


Università degli Studi di Napoli Federico II

Ph.D. Program in

Information Technology and Electrical Engineering

XXXVII Cycle

Thesis for the Degree of Doctor of Philosophy

Assessing and improving isolation in real-time cloud platforms

by

Giorgio Farina

Advisor: Prof. Marcello Cinque

Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

Assessing and improving isolation in real-time cloud platforms

Ph.D. Thesis presented
for the fulfillment of the Degree of Doctor of Philosophy
in Information Technology and Electrical Engineering
by

Giorgio Farina

October 2024

Approved as to style and content by

_____

Prof. Marcello Cinque, Advisor

Università degli Studi di Napoli Federico II
Ph.D. Program in Information Technology and Electrical Engineering
XXXVII cycle - Chairman: Prof. Stefano Russo

http://itee.dieti.unina.it

Candidate's declaration

I hereby declare that this thesis submitted to obtain the academic degree of Philosophiæ Doctor (Ph.D.) in Information Technology and Electrical Engineering is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.
Parts of this dissertation have been published in international journals and/or conference articles (see list of the author's publications at the end of the thesis).

Napoli, June 12, 2025

_____

Giorgio Farina

Abstract

Real-time cloud foresees the porting of critical applications to cloud platforms, in order to enable cloud benefits for critical scenarios, such as ease of usability and reconfiguration, as well as higher scalability and availability. However, this vision has been limited by several threats that live within cloud platforms, mostly related to the sharing of resources.

In order to support real-time cloud vision, this thesis focuses on two key limitations of the emerging real-time cloud paradigm: memory access isolation and failure isolation across execution environments hosted on the same hardware platform.

Regarding memory isolation, this thesis addressed the challenge of evaluating a black-box hardware controller, called "Memory Bandwidth Allocation", available on new Intel server processors, by developing a workload capable of bypassing the regulation shown by generic state-of-the-art workloads by over 50%, warning the community about the risks of adopting generic workloads for specialized controllers. To improve bandwidth utilization, this thesis also provided empirical evidence that memory queue occupancy can detect the interference degree, i.e., the number of memory contenders, within one-fifth of the regulation period.

In terms of failure isolation, this thesis introduced a novel record and replay method to assess hypervisor robustness without manual effort, focusing on the shared, privileged software layer running across customers. This thesis implemented a proof-of-concept tool called IRIS within the Xen hypervisor. The results demonstrated that IRIS could generate new test cases, reaching valid hypervisor states with significant time improvements (from 42.5% to 99.6%) and high accuracy, with code coverage fitting between 92.1% and 100% compared to the recorded execution.

Keywords: real-time, cloud computing, isolation, hypervisor security

Sintesi in lingua italiana

Il paradigma Real-time cloud prevede il porting di applicazioni critiche su piattaforme cloud per abilitare i vantaggi del cloud in scenari critici, come facilità d'uso, riconfigurabilità, maggiore scalabilità e disponibilità. Tuttavia, questa visione è limitata da diverse minacce intrinseche alle piattaforme cloud, in particolare legate alla condivisione delle risorse.

Per supportare la visione del real-time cloud, questa tesi si concentra su due limitazioni principali di questo emergente paradigma: l'isolamento dell'accesso alla memoria e l'isolamento dei guasti tra ambienti di esecuzione di clienti co-localizzati.

In merito all'isolamento della memoria, questa tesi affronta la sfida di valutare il controller hardware "Memory Bandwidth Allocation" presente nei nuovi processori server Intel, sviluppando un carico di lavoro in grado di superare di oltre il 50% la regolazione mostrata dai carichi di lavoro generici adottati dallo stato dell'arte, evidenziando i rischi dell'adozione di workload generici per controller specializzati. Per migliorare l'utilizzo della larghezza di banda, questa tesi fornisce inoltre evidenza empirica che l'occupazione della coda di memoria può rilevare il grado di interferenza entro un quinto del periodo di regolazione.

In termini di isolamento dei guasti, questa tesi introduce un nuovo metodo di record e replay per valutare la robustezza dell'hypervisor, il software layer privilegiato e condiviso tra gli ambienti di esecuzione. È stato implementato un tool proof-of-concept chiamato IRIS all'interno dell'hypervisor Xen, dimostrando che IRIS è in grado di generare nuovi casi di test, raggiungendo stati validi dell'hypervisor con significativi miglioramenti nei tempi (dal 42,5% al 99,6%) e un'elevata accuratezza, con una copertura del codice compresa tra il 92,1% e il 100% rispetto all'esecuzione registrata.

Parole chiave: real-time, cloud, isolamento, hypervisor security.

Acknowledgements

The research presented in this dissertation has been supported by "Consorzio Interuniversitario Nazionale per l'Informatica" (CINI) and by the Meditech Competence Center under the AID4TRAIN Project (I65F21001010005).

Contents

List of Acronyms

The following acronyms are used throughout the thesis.

LLC	Last Level Cache
MMIO	Memory Mapped I/O
PIO	Port I/O
WCET	Worst Case Execution Time
MBA	Memory Bandwidth Allocation
CAT	Cache Allocation Technology
COTS	Commercial Off-The-Shelf
RPQ	Read Pending Queue
IMC	Integrated Memory Controller
PMON	Performance Monitoring
MSR	Model Specific Register
DCLK	DRAM Clock
CAS	Column Address Strobe

RDT	Resource Director Technology
CLOS	CLass Of Service
CDP	Code and Data Prioritization
WB	Write-Back
DRAM	Dynamic Random Access Memory
MPAM	Memory System Resource Partitioning and Monitoring
CMT	Cache Monitoring Technology
MBM	Memory Bandwidth Monitoring
PMC	Performance Monitoring Counters
PCI	Peripheral Component Interconnect
RMID	Resource Monitoring Identifier
HP	Hyperthreading
SMT	Simultanuous Multi-threading
VM	Virtual Machine
VMM	Virtual Machine Monitor
VMCS	Virtual Machine Control Structure
GPR	General Purpose Registers

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing presents the potential to improve efficiency and resource utilization, provide ease of reusability and reconfiguration, as well as higher scalability and availability together with the reduction in maintenance [29]. In addition, it allows industries to reduce their carbon footprint as a result of a reduction in resource over-provisioning. Safety-critical industries can also benefit from cloud computing as it opens a new market segment: safety-critical applications as cloud-based services. EU initiatives and projects such as Digitale Schiene Deutschland [102, 129] and SECREDAS [47] have looked into cloud computing for hosting safety-relevant railway applications. Existing works (e.g. [30]) have also proposed virtualized railway signalling as a cloud-based service. TransVital [128] and DS3 [122] are up-comping platforms from Thales and Siemens to support safety-critical railway applications, such as interlocking and radio block center, in a SIL4 Cloud [56]. The automotive industry is moving towards software-defined vehicles and is exploring Cloud-native as the design paradigm [121]. The Scalable Open Architecture for Embedded Edge (SOAFEE) project [103] is an example of collaboration by automakers, semiconductor suppliers, open source and independent software vendors, and cloud technology leaders (e.g., ARM, AWS, Bosch, Cariad, Continental, and RedHat) to deliver a cloud-native architecture that aims to support mixed-criticality automotive applications. Eker et al. [37] from Ericsson highlights the need and requirements of real-time cloud computing in industry 4.0 use cases such as smart manufacturing.

However, the *isolation* properties of hardware virtualization [111] are a key factor for porting critical applications in the cloud. Isolation properties play a key role in functional safety standards (e.g., DO178C for avionic [113], ISO 26262 for automotive [68], etc.), which recommend providing evidence on temporal, memory, and fault isolation among applications, sharing the same computing infrastructure. The violation of such properties can lead to catastrophic consequences [79].

1.1 Hardware Virtualization and Isolation

Hardware virtualization allows the sharing of the hardware resources of a computer among multiple *virtual* computers, called Virtual Machines (VMs) or *guests*. *Hardware-assisted virtualization* takes advantage of CPU virtualization technologies (e.g., Intel VT-x [96], AMD-V [11], ARM VHE [32]) and other hardware features (e.g., Intel Resource Director Technology [64]) to implement *hardware virtualization*. CPU virtualization technologies introduce a new (and higher) privilege level for the hypervisor, i.e., the Trusted Computing Base (TCB). This way, developers can implement *virtual CPU (vCPU)* abstractions that can run a guest OS at a lower privilege level than the hypervisor. In addition to providing new emulated hardware resources (i.e., device emulation), the hypervisor layer can be used for several purposes, e.g., the hypervisor may trap resource accesses to increase the granularity of resource isolation between the VMs, to optimize the resource utilization, or to make VM introspection. A smaller TCB is generally easier to review or possibly verify, and it is assumed to have fewer vulnerabilities/bugs, so the software complexity of the hypervisor represents a good qualitative metric to measure its isolation robustness.

1.2 Trade-offs in Hardware Virtualization Solutions

Hardware virtualization solutions differ in the *granularity of resource isolation* they provide to VMs and the tradeoffs they make between *hypervisor software complexity* and *resource utilization*. For instance, consider a virtualization solution with a specific granularity of memory isolation:

the user can allocate a fixed, static memory space to the VM (i.e., the VM's memory is not subject to second-level page translation faults, ensuring predictability), but without defined guarantees for the VM's memory bandwidth. Once the granularity is defined, the solution can be implemented in various ways. One approach is to divide the physical memory address space among the virtual address spaces of all VMs. This static partitioning minimizes hypervisor intervention, thus reducing its software complexity, but at the cost of low resource utilization. After memory is allocated to the VMs using second-level page translation, the hypervisor no longer needs to manage memory allocation (no hypervisor intervention). However, this strategy limits the number of possible VMs and leads to low resource utilization, as one VM may require more memory than allocated, forcing it to swap pages to disk, while another VM may underutilize its allocated memory. To improve resource utilization, the virtualization solution could adopt a mixed strategy: static partitioning for critical VMs and dynamic allocation for non-critical VMs. However, dynamic memory allocation increases hypervisor complexity (hypervisor intervention), requiring techniques such as page swapping, deduplication [86], and ballooning [93, 61].

1.3 Threats and Challenges in Real-Time Cloud

Real-time cloud aims to offer to the user high granularity about resource isolation for critical tasks (e.g., providing guarantees about the memory bandwidth, the allocated cache, the IO bandwidth), safeguarding resource utilization and flexibility. However, ensuring strong isolation properties for critical VMs is a challenge, as non-critical VMs hosted on the same hardware platform could potentially interfere with critical VMs, either intentionally or unintentionally.

In this scenario, to break the isolation properties of a critical VM, a non-critical VM may interfere with the critical VM via two primary attack vectors: shared hardware resources (e.g., shared cache, memory bandwidth, etc.) and the software layer shared across the VMs (e.g., the hypervisor). The VM may target the shared hardware resources via eviction-based and contention-based attacks, causing timing failures or exfiltrating sensitive data from the critical VM [145, 105]. To defend

against these attacks, several solutions (software and hardware) propose to partition the shared resources. In that case, the attacker may target the correctness of the regulation/partitioning mechanisms (software or hardware), e.g., attempting to find workloads capable of bypassing the guarantees provided by the current virtualization solution, such as breaking memory bandwidth or cache allocation guarantees. Additionally, the attacker may choose to target the robustness of the hypervisor, the privileged software layer shared between the VMs, by targeting its handling of *sensitive* instructions (e.g., page table updates, interrupt handling, device access, etc.) [100, 99, 98]. Testing the robustness of the current virtualization solutions against these two attack vectors represents a significant open challenge for fostering the adoption of cloud platforms in critical industrial domains. In this thesis, we focused on the following two main concerns.

First, recent cloud regulation mechanisms rely on new hardware features to apply resource partitioning (e.g., Intel Resource Director Technology [64]) provided by vendors as black-box solutions (e.g., Intel, AMD) with limited specification and testing support. Testing the predictability of these features requires ad-hoc workloads designed to stress specific microarchitectural aspects. Existing works primarily focus on cache allocation technology [145], but there is less focus on the predictability of memory bandwidth allocation technologies unless Sohal et al.[125] that rely their evaluation on generic workloads (e.g., DRAM BOMB), without considering the underlying microarchitecture. Second, cloud virtualization solutions often include optimizations that increase hypervisor intervention, complicating the process of reaching sensitive VM states (e.g., setting a new physical CPU state) that require hypervisor intervention. Without deep knowledge of the underlying hardware, attempts to reach valid VM states using random input sequences may cause frequent test VM crashes [161], significantly impacting testing efficiency. Fuzz testing has proven to be effective for identifying isolation vulnerabilities in complex software systems, including hypervisors [161]. The common fuzzing loop involves submitting sequences of VM operations (or reverting to a VM snapshot) to reach a specific VM state, followed by fuzzing input submission to the hypervisor from that state. However, existing hypervisor fuzzers often fall short for the following reasons: (i) They primarily target

I/O virtualization rather than CPU virtualization [115, 95]. (ii) They face challenges in generating valid VM states and creating effective seeds for fuzzing [46, 146].

1.4 Contributions and Major results of this Thesis.

This thesis makes the following contributions.

1.4.1 Memory access isolation

- We first provide a new workload (i.e., Exclusive DRAM Bomb), specific for the new exclusive Intel microarchitecture, capable of triggering the worst execution path for evaluating the indirect limitation enforced by the Intel Memory Bandwidth Allocation (MBA).
- We design a new detection/regulation approach based on the monitoring of memory queue occupancy to provide memory bandwidth guarantees to a critical task while safeguarding resource utilization for the non-critical tasks.

Key findings on a thorough experimental analysis reveal:

- Our workload DRAM-Exclusive Bomb can bypass the regulations shown by the DRAM-Bomb workload [125] over the 50 % in the worst case, warning the community about the adoption of generic workloads to assess black box special-purpose hardware controllers.
- The queue occupancy enables the detection of the number of co-accessing cores. The detection time is at most one-fifth of the regulation period in the worst case when the memory throughput is fully consumed.

1.4.2 Secure CPU Virtualization

- We propose IRIS [1], a framework to *record* (learn) sequences of inputs (i.e., VM seeds) from the real guest execution (e.g., OS boot), *replay*

[1] From Greek mythology, a messenger of the gods and the personification of the rainbow, which connects the world of the gods with humanity

them as-is to reach valid and complex VM states and finally use them as valid seed for a fuzzing. A preliminary implementation of a proof-of-concept fuzzer, built upon the findings above, demonstrates the feasibility of using the proposed approach as a first step for assessing hardware-assisted virtualization.

From our experiments, we obtain the following findings:

- The *accuracy* of *IRIS* to automatically generate (record) seeds to reproduce real VM behaviors. The results show a fitting of code coverage ranging between 92.1% and 100% in our settings, compared to real guest execution.
- The *efficiency* of *IRIS* to replay recorded seeds to reach a valid VM state, with a time improvement (compared to the real guest execution.

1.5 Overview of the thesis structure

The rest of the thesis is structured as follows. Chapter 2 provides an overview of the problems and threats we have in the real-time cloud paradigm, including the two issues addressed in this thesis. Chapter 3 provides the technical background required to understand peculiar aspects of the work, especially those presented in Chapter 5 and Chapter 6. We introduce the architectures of the tools, designed to achieve our goals, in Chapter 4, and the respective implementations in Chapter 5. In Chapter 6, we present the research questions of this thesis, and we respond to them by introducing the adopted experiments and methodology. In Chapter 7, we present the recent results in output by a work-in-progress implementation, highlighting the next planned efforts and future directions. Chapter 8 presents the works related to the problems addressed in this thesis. Finally, Chapter 9 concludes the thesis.

Chapter 2

Problem statement

We provide in Section 2.1 an overview of the existing threats living within cloud platforms that limit the adoption of this technology for a critical scenario. Afterwards, Section 2.2 introduces the two specific problems addressed in this thesis, positioning them within the classification presented in Section 2.1.

2.1 Threats in Real-Time Cloud

We assume the goals of a malicious cloud customer are (i) to leak or modify sensitive information and/or (ii) to deny, or degrade the performance, of services of other cloud customers that share the same physical hardware machine, to cause damage to their business, or the cloud vendor itself. Below, we provide an overview of the main existing attacks that regard the attack vectors addressed in this thesis. Instead, we do not provide an overview of other attack vectors that are out of scope, e.g., instruction timing attacks or Virtual Machine (VM)/Virtual Machine Monitor (VMM) fingerprints for virtualization detection [78, 24], VM-based rootkits [75], and attacks on trusted execution and secured boot technologies [139].

2.1.1 A1: Microarchitectural attacks via shared resources.

An adversary VM may exploit unintended effects (e.g., timing variations) in accessing a particular shared resource to surreptitiously exfiltrate data (in the *covert channel case*), infer a VM victim program's secrets (in the *side channel case*) or cause the performance degradation on a VM victim (in the *performance degradation case*), We extend the classification of Paccagnella et al. [105] to include also DoS attacks [94, 155, 80], which assume more relevance in RT-Cloud. In real-time cloud environments (RT-Cloud), DoS attacks not only degrade user performance, increase costs, and damage the cloud provider's reputation, but they can also cause timing failures (e.g., missed deadlines). Such failures may propagate through the system, ultimately compromising the safety of individuals and leading to potentially catastrophic consequences [152, 79]. The presented classification groups the attacks according to the microarchitectural resource that they exploit and on the degree of concurrency (referred also as *leakage channel*) they rely on. Finally, we discuss the current solutions (defenses) to mitigate the attacks.

Limited and shared storage The root cause of these attacks is the limited storage space of the shared microarchitectural resource, e.g., L1 data [83, 104, 110] and instruction caches [2, 3, 156], the TLB [53], the branch target buffer (BTB) [39, 40], and the Last Level Cache (LLC)[33, 50, 54, 55, 67, 70, 85, 90, 117, 148, 157]. The attackers leverage the limited storage capabilities to build eviction-based attacks (also referred to as persistent-or residual state) that are stateful. The adversary actively brings the microarchitectural resource into a known state and lets the victim execute. In this state, the nominal execution of the victim is affected (performance degradation attacks) [19, 23, 77, 88, 153]. In the case of side channels, the attacker, after the victim's execution, checks the state of the shared resource again to learn secrets about the victim's execution, as the attacker assumes the side effects of the victim's execution leave a footprint that is not undone when the victim code completes.

Limited and shared bandwidth The root cause of these attacks is the limited bandwidth capacity of the shared resource, e.g., functional units[136], execution ports[10, 20, 52], cache banks [149], the memory bus [158, 31, 105, 133] and random number generators[38]. The attackers aim to set up contention-based attacks (also known as transient states) which

are stateless. The adversary accesses the shared resource contended with the victim to generate interference (performance degradation) [109, 5, 72]. In the case of side channels, the adversary passively monitors the latency to access the shared resource and uses variations in this latency to infer secrets about the victim's execution. Instead of the state-full attacks, the side effects of the victim's execution are only visible while the victim is executing. For instance, Dai at al.[31] first showed that through mesh contention they can reach a covert channel with channel capacity over 1.5 Mbps, and then they demonstrate that interconnect side-channel attacks can be used to leak keys from vulnerable cryptographic implementations.

Degree of concurrency These attacks rely on specific CPU availability conditions to make the attack feasible. As [105], we classify the attacks as either relying on *preemptive scheduling, Simultanuous Multithreading (SMT) or multicore techniques*. In the *Preemptive scheduling* approaches, also referred to as *time-sliced* approaches, the victim and the attacker share a core. In these attacks, the victim and the attacker run on the same core and their execution is interleaved. The `SMT` approaches rely on the victim and the attacker executing on the same core in parallel (concurrently). Finally, in the `Multicore` approaches, the victim and the attacker run on separate cores.

Defenses The main defenses to defeat microarchitectural attacks on shared resources are the disabling of sharing and shared resource partitioning. The most straightforward approach to block interference attacks (i.e., unpredictability) is to disable the sharing of the microarchitectural component on which it relies. For example, attacks that rely on SMT can be thwarted by disabling SMT, which is an increasingly common practice for both cloud providers and end users [14]. Other approaches propose to partition the shared resource across customers, to ensure that the resources of the victim cannot be interfered and monitored by the attacker [74, 84, 126, 135, 160]. For example, Liu et al. [84] present a defence to multicore cache attacks that use Intel CAT [64] to load sensitive victim cache lines in a secure LLC partition where the attacker cannot evict them. Specific for the state-full attacks that rely on preemptive scheduling and SMT, one solution consists of erasing the victim's footprint from the microarchitectural state across context switching.

2.1.2 A2: Attacks via Virtual Machine Monitor

Cloud providers rely on hardware-assisted virtualization to provide isolation across distinct execution environments running on the same hardware platform. They deploy distinct software components in different execution environments, called VM that abstract a physical computer. However, a privileged software, called VMM (also known as a hypervisor), may interrupt the execution of these VMs, to emulate missing hardware features, or improve and control the utilization of existing underlying hardware resources. Since this software is shared across the VMs and part of its code runs at a more privileged level, bugs within the VMM may affect all the hosted VMs: a VM may intentionally or unintentionally trigger bugs in this layer, with the risk of causing problems to the co-located VM. Below, we provide an overview of the existing attack interfaces, especially those related to an unmodified guest (i.e., the guest OS is not aware of running within a virtual environment, so the virtual environment is faithful to the real hardware interface).

Virtual Device. Virtualizing a device means providing the guest VM with an abstraction of the real physical interface, such as Memory Mapped I/O (MMIO) or Port I/O (PIO), corresponding to the device. Accessing this peripheral interface triggers hypervisor intervention (VM exit) in the most privileged mode. The hypervisor then handles the device emulation in the userspace. Vulnerabilities in device emulation can have a different impact depending on their effect on the other VMs and the hypervisor, e.g., if the process has not restricted access to the hypervisor kernel, the malicious VM can exploit the exploited process to run rootkits.

CPU Virtualization. CPU virtualization requires the emulation of specific processor behaviors and memory layouts of the guest. During the hypervisor intervention, e.g., when the guest accesses an MMIO area of a virtual device, the hypervisor updates the guest CPU's state before resuming the VM. This process mirrors how a real CPU's state would be updated under similar conditions. However, emulating CPU behaviors can be prone to errors [49, 12], especially with complex CISC processors that count several CPU states. Additionally, CPU emulation requires running in the most privileged mode, which introduces potential vulnerabilities that could compromise the entire system, with the major risk of privilege escalation [49].

2.2 Addressed problems

This work addresses the following aspects of these attack vectors.

2.2.1 Limited and shared memory bandwidth in Multi-Core platforms

To port critical applications in cloud, we need to provide temporal isolation to these applications when scheduled on a physical core, even in case of performance degradation attacks via shared resources (Attack Vector A1). Existing cloud virtualization environments, however, provide limited isolation amongst the applications [47]. Popular real-time hypervisors focus mainly on CPU virtualization, adopting either hierarchical scheduling [142] or CPU partitioning [123], and memory space virtualization among the applications. Several works [127, 137] consolidated space partitioning techniques for the LLC, even when the mapping address function is more complicated than a module [43]. In addition, recent works explored hardware features, such as the Intel Cache Allocation Technology[124, 144, 51]. However, once the partitioning of the LLC is implemented, memory bandwidth regulation is still necessary to limit memory access contention. On this line, resource limitation approaches can be used to find tight upper bounds for the memory fetching phase of a critical task running on a physical core. This is done by limiting the memory accesses of non-critical tasks running on other cores, within a regulation period. The most popular software-based implementation of memory bandwidth limitation is MemGuard[152]. It monitors the memory requests (or cache misses) from each core. It applies a throttling policy to prevent memory accesses from cores that have depleted the assigned memory budget for that regulation period. However, the throttling policy of the approach involves stalling the core execution to prevent memory access. Such a throttling policy is unsuitable for cloud VM as a core cannot continue its execution within the private context, such as private caches and functional units, reducing the private resources utilization. Luckily, modern processor manufacturers introduced support for Memory bandwidth Allocation on the modern Intel Xeon Scalable processors[62], which is a hardware controller to limit the memory bandwidth of non-critical cores. MBA delays the requests going to the high-speed interconnect from a core's private context ; however,

MBA is a black box controller and the allocated memory bandwidth is an indirect consequence of the introduced delay, so, it needs a systematic evaluation.

Contributions. We provide a microarchitectural-specific workload to obtain a deterministic regulation from MBA1.0 for memory bandwidth partitioning. We show that our synthetic workloads can break the regulation showed by previous evaluations based on generic synthetic workloads [143, 124]. Afterwards, we evaluate the memory queue occupancy as a detector for interference, to activate the regulation only when necessary, safeguarding the performance of non-sensitive applications.

2.2.2 Secure CPU Virtualization

Trap-and-emulate is a key feature in virtualization. Hypervisors may trap for a variety of reasons, such as handling accesses to memory-mapped I/O (MMIO) regions to emulate virtual devices or emulating specific instructions and hardware features that are not natively supported. Each time the hypervisor traps the execution of a guest OS (i.e., VM exit), it can access critical information about the VM's (VM) state to address the cause of the exit, such as fetching and decoding the trapped instruction. Following this, the hypervisor must correctly update the virtual CPU (vCPU) of the guest OS before resuming execution (i.e., VM entry). These operations in handling VM exits are prone to errors, as highlighted in prior work [12, 49], especially in cloud environments where complex CPU architectures are used, e.g., the Intel Reference Manual counts more than thousands of pages [64]. Due to the inherent complexity of these operations, it is paramount to accurately test these exit conditions to ensure that untrusted guest OSs cannot exploit them to jeopardize the execution of other co-located VMs.

However, the conditions that trigger VM exits vary across hypervisors and can be difficult to reproduce, often requiring a deep understanding of the underlying hardware, i.e., the knowledge of an OS. For instance, a frequent source of VM exits occurs when a guest OS transitions between different CPU modes, such as switching from real mode to protected mode. Enabling protected mode, as described in Section 9.9.1 of [66], requires setting up an artificial guest workload that includes several complex steps, such as clearing interrupts and setting up the Global Descriptor Table

(GDT). These operations ask to be executed in a strict sequence; any deviation can result in hardware exceptions.

Our key insight is that instead of manually reproducing these complex guest workloads [49], we can replicate existing behaviors. For instance, an OS already executes the necessary low-level instructions to transition to protected mode during the boot process, so why not capture and learn from these naturally occurring behaviors?

Contributions. We propose a record-and-replay framework that enables the efficient recording (learning) of hypervisor behaviors during the execution of guest workloads. Additionally, our replay mechanism can submit recorded hypervisor behaviors as a series of VM exits, eliminating the need to re-execute guest workloads. The framework also supports injecting crafted VM exits to test corner cases, facilitating fuzzing for edge cases in hypervisor handling.

Chapter 3

Background

We provide a technical overview of the technologies adopted and exploited in this thesis. Section 3.1 introduces the Intel Monitoring/Regulation Capabilities available on the new Intel server processors. This section is useful to understand the part of the thesis regarding memory access isolation. Instead, Section 3.2 and Section 3.3 provide sufficient technical details of Intel virtualization extension to understand the design and implementation of the hypervisor behavior record and replay.

3.1 Overview of the Intel Monitoring and Regulation Capabilities

We provide a summary of the recent Intel hardware features that support temporal and spatial isolation in the memory hierarchy.

Given a *Technology*, Table 3.1 describes the *Action* provided. The first half of the table includes monitoring capabilities, while the second half includes regulation capabilities. The *Action* is performed from a core and *targets* a set of cores (*Multi-core*), or the core where the action is performed (*Per-core*), or any cores (*Un-core*, meaning "unaware of the core"). The programming hardware interface is specified in *Interface* (i.e., by Model Specific Register (MSR) or Peripheral Component Interconnect (PCI)). The column *Int.* specifies if the monitoring feature supports the setup of a threshold on the monitoring events and thereby the generation of asynchronous events once passed the threshold. All these *Actions* can be

performed online.

For further clarity, we explain the first two rows of the table. The core performs "set Resource Monitoring Identifier (RMID)", which means targeting itself, as the action is *Per-core*, with a RMID. While in the second row, the core performs "read RMID performance". In this case, it reads the monitored performance of all cores that are targeted with that RMID (*Multi-core*). It is not possible to set a threshold on the monitored events to cause an interruption ("no Int." in Table 3.1).

Table 3.1. Summary of Intel's monitoring and regulation technologies

Technologies	Properties			
	Action	Interface	Targeting	Int.
Detection/Monitoring capabilities				
RDT	set RMID	MSR	Per-core	no
CMT	read RMID perf.	MSR	Multi-core	no
MBM	read RMID perf.	MSR	Multi-core	no
Per-core PMC	read PMC	MSR	Per-core	yes
Un-core PMC	read PMC	MSR/ PCI	Un-core	yes
Regulation capabilities				
RDT	set CLOS	MSR	Per-core	-
CAT	set ways to CLOS	MSR	Multi-core	-
MBA	set delay to CLOS	MSR	Multi-core	-

Per-core Performance Monitoring Counters. The per-core Performance Monitoring Counters (PMC) [1] count the performance monitoring events from each core's perspective. For instance, one core can configure one PMC to count its "LLC miss" [2]. The counter can be configured to generate an interrupt once it has reached a given number of events. Generally, each physical core supports eight PMCs. In the case of Hyperthreading (HP), the PMC are divided between the virtual cores hosted in the physical core.

Un-core Performance Monitoring Counters. The un-core PMC are local counters of the shared resources. They are generally unaware of the target core and they can be read from any core. The un-core ar-

[1] Note Intel refers to the Per-core and Un-core PMC as "PMC" and "un-core PMC" respectively.

[2] The "LLC miss" event is used by state-of-the-art approaches such as Memguard [152].

chitecture has several boxes enclosing single physical components such as the high-speed interconnect, the memory controller, and the shared LLC. These boxes include Performance Monitoring (PMON) local blocks. Each PMON local block includes the control, status, and data for the counters. The programming of PMON local blocks is through MSR or PCI interface. In addition, un-core monitoring includes the PMON global block, which (i) allows taking some actions on the PMON local blocks through a unique global structure. Furthermore, (ii) once the counters overflow, the PMON global block also allows interrupting a specified subset of physical cores to run a handler. Molka et al. [92] discuss the accuracy of un-core monitoring in detecting memory bandwidth utilization. Un-core monitoring is adopted both in real-time and security systems for reverse engineering [58] [91] or in green computing to estimate and reduce the power consumption [81] [13].

Integrated Memory Controller boxes in our architecture. The Intel Xeon Scalable Processor un-core monitoring [65] has two memory controllers, which are respectively enclosed in two distinct Integrated Memory Controller (IMC) boxes. Each IMC box includes respectively one PMON local block for each Dynamic Random Access Memory (DRAM) channel. The IMC boxes are exposed through the PCI interface. One PMON local block is composed of four counters and one fixed counter. The fixed counter only tracks the number of DRAM Clock (DCLK), which never changes frequency; therefore, it is a good measure of the wall clock. Instead, the other counters can monitor several events, such as DRAM refresh cycles, DRAM Column Address Strobe (CAS) commands, and the occupancy of one DRAM channel. Memory writes (CAS write commands), in particular, can only be monitored globally[3] when the LLC is shared because other cores can also cause WBs.

The Read Pending Queue occupancy event. RPQ occupancy can be monitored via the IMC PMON box un-core monitoring in the Intel Xeon Scalable processor [65]. The RPQ is used to schedule reads out to the memory controller and to track the requests. Requests are

[3]For instance, the LLC misses can be monitored as a per-core event because the LLC miss is caused by the physical core that issued the request. Instead, the LLC Write-Back (WB) can be caused by other physical cores that demand other data, and thereby it is not provided as a per-core event.

allocated into the RPQ soon after they enter the memory controller, and need credits for an entry in this buffer before being sent to the IMC. The RPQ occupancy event accumulates the occupancy of the RPQ each DRAM clock[4] (see Section 2.3.7 of [65]). In particular, the setup includes the definition of a threshold on the occupancy; the counter counts the number of cycles that the occupancy (the number of queued requests) is higher than this threshold. For instance, if we set the threshold to one, the counter is increased every time the queue holds at least one request during one cycle. One entry is deallocated from the RPQ once the memory operation has been issued to the memory; hence, by setting the threshold to one, this counter provides the busy time (in cycles) of the queue.

Resource Director Technology. The 2^{nd} Generation of Intel Xeon Scalable Processors [62] is based on the Intel Cascade-Lake microarchitecture. The considered micro-architecture and all the new generations of Intel Xeon Scalable processors support temporal and spatial isolation at the hardware level through Resource Director Technology (RDT) [64]. The MSR allows interfacing with RDT. RDT programming includes associating spatial and temporal properties with a CLass Of Service (CLOS) and binding the CLOS to a virtual core. At the same time, RDT programming allows monitoring of the spatial and temporal consumption of shared resources by binding a RMID to a virtual core. Hence, each virtual core can be associated with CLOS and an RMID. One virtual core cannot change the class of service (or RMID) of another core. However, any virtual core can configure the properties of any CLOS and can read the monitored performance associated with an RMID. Using RMID, it is also possible to monitor the cache space occupancy (Cache Monitoring Technology (CMT)) and the memory bandwidth (Memory Bandwidth Monitoring (MBM)). They can be read via polling (i.e., in a synchronous way). No thresholds can be set to enable asynchronous handling. The RDT allocation features associated with CLOS enable the partitioning of shared resources between the cores. While Cache Allocation Technology (CAT) and Code and Data Prioritization (CDP) focus on the LLC spatial partitioning, MBA addresses the temporal memory contention.

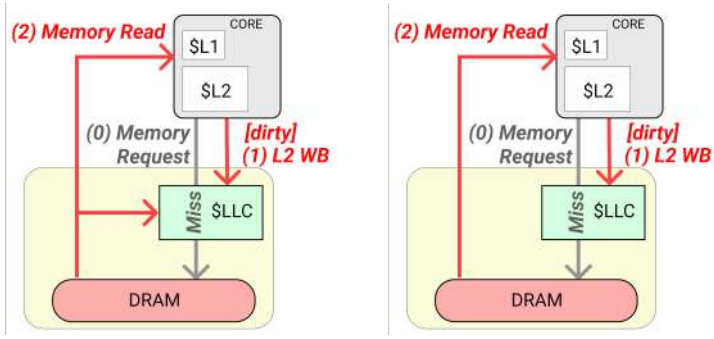
Memory Bandwidth Allocation 1.0. MBA inserts a configurable delay between requests going to the high-speed Interconnect (L3 Inter-

[4]One DRAM clock corresponds to half of the DRAM Speed

connect) from a core's private context (L2 cache). The granularity of MBA control is linear, starting at 10% and going up to 100% in increments of 10%. Through the delays, we can apply an indirect limitation in memory access without impacting the use of private resources such as private caches and functional units. An MBA delay value is associated with a CLOS. This binding can be done in any core. For instance, we can associate the critical and non-critical cores with distinct classes of service, $CLOS_X$, and $CLOS_Y$, respectively. Once the critical cores access the memory, we can limit the memory accesses of non-critical cores, changing the delay of $CLOS_Y$. Note two virtual cores share one physical core. If two virtual cores (hardware threads) of the same physical core have different delays, the larger delay will override the smaller one. In our micro-architecture, disabling HP in the experiments, we have a side effect [62]; in particular, the non-active hardware threads are assigned to the CLOS0 by default even if they are disabled. Hence, CLOS0 cannot be used. Otherwise, the hidden delay value of the non-active virtual core (the delay value associated with the CLOS0) overrides the set MBA delay value of the active virtual cores on the same physical core when the first delay is higher than the second one.

Memory Bandwidth Allocation 2.0, 3.0. The next generation MBA applies some enhancements to improve efficiency and accuracy in throttling, along with providing increased system throughput [63]. Rather than a strict bandwidth control mechanism, a dynamic hardware controller is implemented, which can react to changing bandwidth conditions at the microsecond level. Even if the MBA 2.0, 3.0 is located again between requests going to the high-speed Interconnect (L3 Interconnect) from a core's private context (L2 cache), the new hardware controller includes a feedback mechanism that tracks actual DRAM bandwidth to throttle only requests to the DRAM, avoiding to throttle requests that hit the last level cache. In addition, they provide (in some architectures) the possibility to calibrate the delay from the BIOS. MBA 3.0 adds the capability to associate the throttling to a specific virtual core, which means that two virtual cores that share the same physical core can have their private levels of throttling. Our microarchitecture supports MBA 1.0 and thereby our analysis is limited to the MBA 1.0, even though we believe its insights might be significant for the evaluation of the other controllers.

Exclusive Last Level Cache. The Cascade-Lake and the newer Ice-Lake micro-architecture inherited from the Skylake-Server micro architecture have an exclusive LLC and a mesh network as a shared interconnect. The property of exclusivity changes the memory traffic patterns in modern Intel processors. Figure 3.1 shows how the inclusive and exclusive architectures handle the LLC misses, respectively. Whenever an LLC miss occurs in inclusive architecture, the new cache line fills both LLC and private cache, while, in exclusive architecture, only the private cache is replenished. The two figures suppose that there is available space in LLC. The different reactions to the LLC miss influence memory read workloads. Suppose we start with empty private caches. Once we fill up all the private caches, we do not have an additional L2 WB in the inclusive architecture because the cache lines are already updated and available in LLC. In the exclusive architecture, instead, there are L2 WBs. Unlike previous works (e.g., [124]), we consider this behavior while evaluating MBA to obtain more accurate results regarding the worst-case interference.



(a) Inclusive Architecture **(b)** Exclusive Architecture

Figure 3.1. Last Level Cache miss handling

3.2 Overview of the Intel virtualization extensions

Virtualization. Virtualization techniques allow running multiple Operating Systems (OSes) on the same hardware. The main software compo-

nent that enables virtualization is the *hypervisor* or *VMM*. It is responsible for creating, managing, and scheduling Virtual Machines (VMs), which represent an abstraction of CPUs, memory areas, and devices of a real machine. Early virtualization techniques were based on *full virtualization*, which completely emulate privileged instructions and I/O operations. To achieve higher performance, *paravirtualization* involves modifying guest OSes (i.e., OSes running as VMs) to directly use the services offered by the hypervisor through the so-called *hypercalls*. In the last years, virtualization evolved to exploit hardware extensions in modern CPUs, to enable full virtualization while improving overall performance [97, 4].

Intel VT-x and VM Life-cycle. Intel VT-x [66], is the target technology we used to implement the record and replay framework. Once the virtualization is enabled (`VMXON` instruction), two operating modes are active. The hypervisor (even called *VMM*) operates in *root mode*, while the guest *VM*s run in *non-root mode*. The latter modes are orthogonal to traditional execution modes (long, protected, and real modes) and to privilege levels (i.e., rings). Running a new VM in *non-root mode* requires allocating and initializing in memory a particular control structure, called *(Virtual Machine Control Structure (VMCS))*, linked to a specific vCPU. The VMCS, except for its first eight bytes, must be read and written by executing dedicated *VMX instructions* called `VMREAD` and `VMWRITE`, otherwise unpredictable failure modes can occur (see Section 24.11.1 in [66]). The VMCS consists of the following areas: *guest-state*, *host-state*, *control fields*, and *VM exit information*. The first two are the most important in the context of our framework and include, respectively, the processor state when the VM is suspended and resumed. Specifically, they include special-purpose registers (e.g., control registers, instruction pointers, etc.).

Figure 3.2 depicts the VM lifecycle. The VMCS is initialized (`VMCLEAR` instruction, step ① in Figure 3.2) during the VM startup and subsequently loaded (`VMPTRLD` instruction, step ② in Figure 3.2). When the VMCS is loaded, its internal hardware state becomes *Active Current Clear*. In this state, the hypervisor can set up the VM, for example, by defining the events and instructions in *non-root mode* that will cause a switch to the *root mode* (i.e., a *VM exit*). Once the setup is completed, the hypervisor can launch the VM (`VMLAUNCH` instruction, step ③ in Figure 3.2). Once this instruction is complete, the VMCS state becomes *Active Cur-*

rent Launched and the Guest VM can run, after switching to non-root mode (called *VM entry*).

During the execution of the VM, the control can pass to the hypervisor every time a *VM exit* occurs, requiring a context switch from non-root to root mode. VM exits can occur for different reasons. Currently, Intel *x86* architecture support 69 *VM exit reasons* (Appendix C, Table 1-c [66]). Most of them are due to the execution of sensitive instructions by the VM, such as `RDMSR`, `WRMSR`, or `CRx ACCESS`. Others include VM events or conditions to be handled by the hypervisor, such as triple fault, interrupts, and I/O port access. Finally, the hypervisor can decide to trap some VM conditions to follow the VM evolution (e.g., VM introspection [57]) or to take scheduling and resource-sharing decisions (e.g., memory deduplication [86]).

The VM exit is a key operation since it can be exploited to compromise the isolation properties of the hypervisor. Hence, we use it as the mean to submit a seed to the hypervisor and to test its operation. Let us analyze in detail the steps occurring from the VM exit up to the successive VM entry (VM resume), including the execution of the *VM exit handler* in the hypervisor (steps ④ and ⑤ in Figure 3.2). The VM exit requires a hardware context switch from non-root to root mode, that entails: (i) to save the physical processor state in the guest-state area of the VMCS (except for general purpose registers (General Purpose Registers (GPR)s), saved in the hypervisor data structure), (ii) to load the new root mode processor state from the host-state area of the VMCS, including also the instruction pointer register (RIP), containing the start address of the VM exit handler. After the context switch, the VM exit handler identifies, from the VMCS, the cause of the exit and appropriately resolves it. More importantly, during the execution, the VM exit handler can access the entire VMCS (`VMREAD`, step ④ in Figure 3.2), hence its control flow depends on VMCS fields. Additionally, the VM exit handler can change the VM state in the VMCS (guest-state area) (`VMWRITE`, step ④ in Figure 3.2). Once the VM is resumed (`VMRESUME` instruction, step ⑤ in Figure 3.2), the new VM state becomes operational on the physical CPU. The VMRESUME performs a new (inverse) hardware context switch, where the processor state is loaded from the guest-state area of the VMCS.

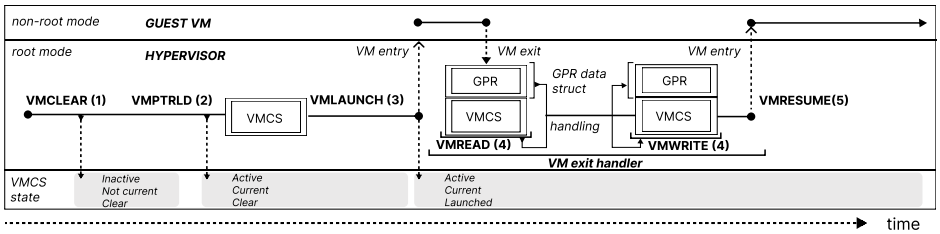


Figure 3.2. Workflow of a virtual machine in VTX

3.3 Overview of Nested Virtualization

Linux KVM incorporates nested virtualization [18] in its software functionalities, which we briefly discuss below.

In nested virtualization, a hypervisor host (i.e., L0) can host a hypervisor guest (i.e., L1), which in turn can host its VMs (i.e., L2). Since Intel VT-x supports a single level of virtualization, nested virtualization requires the software emulation of VMX operations by the hypervisor host, to multiplex the hardware virtualization extensions between multiple hypervisor guests.

As explained before, the hypervisor can take control during the execution of a guest OS on a vCPU. In nested virtualization, the hypervisor guest and its guest OS are decoupled in two distinct VMs (i.e., L1 and L2 VMs) of the hypervisor host (i.e., L0). Only one between L1 and L2 is in execution, abstracting a unique vCPU.

L1 runs as VM in L0 with its VM specifications (VMCS01). L1 is not aware that it is running in guest mode and uses VMX instructions (e.g., VMXON, VMXPTRLD, VMREAD, WMWRITE) to define and launch the specifications for its guest, L2 (VMCS12). However, VMX instructions can only execute successfully in root mode, so L0, running in root mode, traps and emulates the L1 VMX instructions, returning the control to L1. Instead, the processing of VMLAUNCH and VMRESUME is different, as L0 needs to emulate a VMEntry from L1 to L2. These instructions cause, first, a VMexit from L1 to L0 and then, a VMentry from L0 to L2. However, L0 cannot use VMCS12 to execute L2 directly, since the VMCS12 is just a software structure in L0's environment. Thus, L0 runs L2 with a new specification (VMCS02), considering all the specifi-

cations defined in VMCS12 and those defined in VMCS01. For instance, the *control fields* of the VMCS02 result from the merge of VMCS01 and VMCS12, to trap the events at which either L0 or L1 are interested.

Hence, once L2 is in execution (i.e., L1 is suspended), an event can call the intervention of L0, which in turn can decide to handle it by itself (① in Figure 3.3), or forward it to L1 (②a, ③, ②b in Figure 3.3).

The forwarding is enabled by trappable events specified in VMCS12, related to L1. L0 forwards the event to L1 by (in ① of Figure 3.4) copying VMCS02 fields (e.g., the *guest state* and *VM exit information* areas), which are updated by the processor, to the VMCS12 and (② in Figure 3.4) resuming L1. The hypervisor running in L1 believes there was a VMexit directly from L2 to L1 thanks to L0 properly updating VMCS12. The L1 hypervisor handles the event (③ in Figure 3.4) and later resumes L2 (④ in Figure 3.4) by executing `VMLAUNCH` or `VMRESUME`, both of which will be emulated by L0, as explained before.

About memory virtualization, the hypervisor host should emulate the Extended Page Table [18] to the hypervisor target, by which it provides a virtual memory space for its guest (EPT12). However, even in this case, the specification of L0 (EPT01) and L1 (EPT12) are compressed into one (EPT02). The building of EPT02 is on the fly, trapping every first access to a new memory page during L2 execution [18].

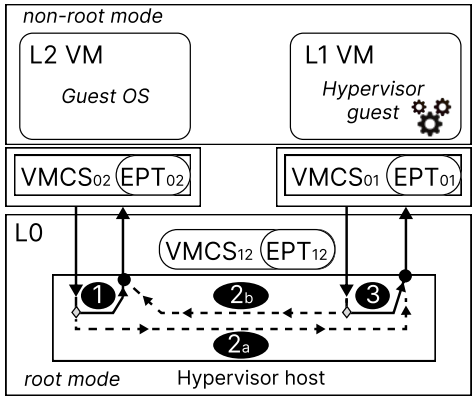


Figure 3.3. VMX multiplexing

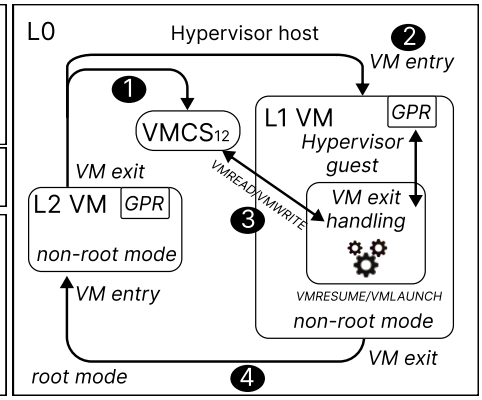


Figure 3.4. VM exit Forwarding

Chapter 4

Design

We introduce the design principles of the tools implemented in this thesis. Section 4.1 describes the two main steps of the process we designed to test the Intel MBA: single-core analysis and multi-core validation. Instead, Section 4.2 focuses on the design of the record and replay of hypervisor behavior. In Section 4.2, we first discuss the design goals of the framework, and then we introduce the assumptions on the hypervisor model made by the current implementations.

4.1 Memory Access Isolation

This section introduces a tool designed to evaluate the regulation mechanisms enforced by Intel Memory Bandwidth Allocation (MBA). We propose a two-step process for testing a black-box hardware memory bandwidth controller: single-core analysis followed by multi-core validation.

In the single-core analysis, the primary goal is to understand the controller's regulation mechanisms and operating modes. This phase allows observation of the target behavior in a controlled environment. According to the Incremental Development and Certification (iD&C) approach, once a reliable model of the controller's behavior is developed, the multi-core analysis seeks to validate this model by reproducing worst-case interference scenarios. This provides empirical evidence of the guarantees enforced by the controller.

4.1.1 Single-core Analysis

To study the controller's behavior, we designed two controlled testbeds, periodic sampling and latency sampling, as illustrated in Figure 4.1 and Figure 4.2. These testbeds aim to simplify the analysis effort, allowing precise observation of controller behavior under varied conditions.

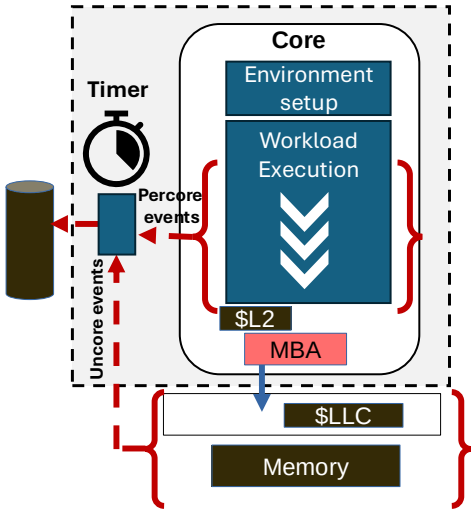


Figure 4.1. Periodic Sampling

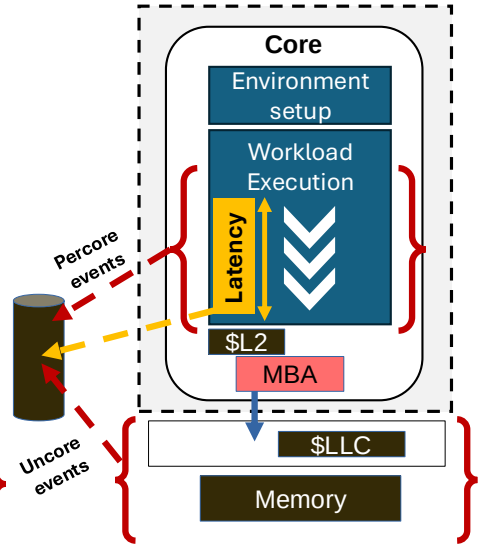


Figure 4.2. Latency sampling

Periodic sampling. This operating mode enables periodic sampling of micro-architectural events during workload execution, with a configurable period we call the regulation period. Before recording begins, the tool allows for MBA delay configuration on the relevant physical cores. This mode is useful for a high-level view of controller behavior, such as observing the effects of regulation on generic memory interference workloads.

Latency sampling. This operating mode enables fine-grained measurement of workload execution time. For instance, the user may specify a workload of a few hundred memory operations, and the tool records micro-architectural events along with execution time. This mode enables a high control degree on the workload. The user controls starting conditions (e.g., no preemptions, empty L2 cache, LLC availability, MBA

enabled) and can run a specified number of instructions to create a targeted traffic pattern. This degree of control is not available in periodic sampling, which inherently provides only a periodic view.

4.1.2 Multi-core validation

The multi-core analysis aims to provide empirical evidence of the guarantees provided to a critical core when interference workloads run on parallel cores. As illustrated in Figure 4.3, this step should support the setup of workloads and environmental conditions for both interfering and protected cores, the configuration of MBA delays on interfering cores, and the tracing of per-core and uncore events across all cores during the protected core's workload execution. Event tracing is essential to provide empirical evidence of the observed guarantees.

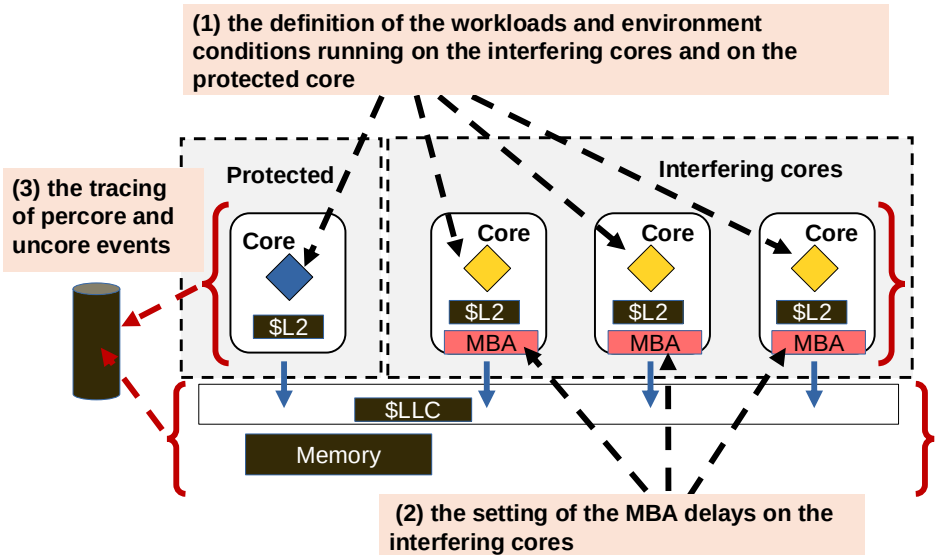


Figure 4.3. Design goals for the multi-core analysis.

4.2 Record and Replay of Hypervisor behavior

4.2.1 Design Goals

Testing hypervisor intervention is challenging for two main reasons: ① Hypervisor intervention is triggered by VM conditions that are implementation-specific. This means that test cases that are relevant for one hypervisor may not be significant for another. ② Reproducing these VM conditions requires a deep understanding of the underlying hardware (e.g., CPU specifications), comparable to the level of knowledge required for Operating System internals.

Instead of manually recreating these complex guest workloads for each hypervisor, our key idea is to replicate existing hypervisor behaviors observed during the nominal execution of guest workloads.

Recording Goals. Figure 4.4 shows the action planned during the recording phase. During the execution of guest workloads within a virtual machine, our recording mechanism should capture key events from every hypervisor intervention (i.e., those triggered by a VM exit) to enable the replay of these observed interventions at a later time.

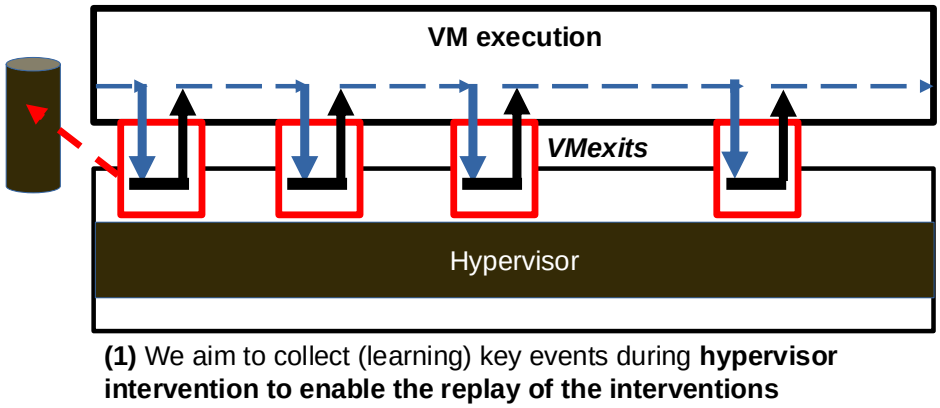


Figure 4.4. Design goals for the recording process.

Replay Goals. Using the events collected during the recording phase, the replay mechanism aims to reproduce hypervisor interventions without executing the actual guest workloads. Avoiding the execution of guest

workloads is crucial for performance, as these workloads may generate only a few VM exits (although significant) over a prolonged time window.

As illustrated in Figure 4.5, we first aim to use the replay of hypervisor interventions to navigate through observed hypervisor states (② in Figure 4.5). Afterwards, the framework should support the injection of crafted VM exits to enable the automatic generation of new test cases without manual effort (③ in Figure 4.5).

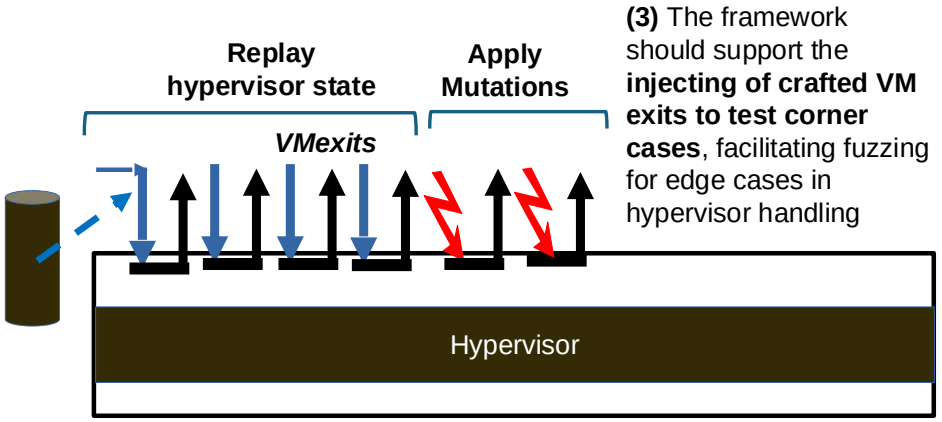


Figure 4.5. Design goals for the replay process

4.2.2 Design alternatives

We explore two design approaches for implementing record and replay mechanisms of hypervisor behavior:

- **On-device design:** This approach extends the target hypervisor to support record and replay features without introducing an additional layer of virtualization. It is beneficial when preserving the hypervisor's real-time performance is critical. We implemented a prototype of this design in IRIS, an on-device record and replay extension for the Xen hypervisor.
- **Transparent design:** This alternative enables record and replay

capabilities without modifying the target hypervisor, to be non-intrusive (in terms of software) and generic. We developed a first work-in-progress version based on this design, called HyRo, as an extension for KVM/x86.

The choice between these two approaches depends on the specific application scenario. For example, the IRIS design may be preferable in scenarios where VM exit performance is crucial, such as estimating worst-case execution times. Instead, if the primary goal is detecting software bugs or studying failure modes, the user might prioritize the non-intrusive nature of the HyRo design. Additionally, a hybrid approach can be considered. For instance, cloud providers may instrument their hypervisors to collect traces in real-time using on-device instrumentation, while offloading these traces to HyRo on an external server for debugging, ensuring that the target hypervisor continues to run services uninterrupted.

Below, we outline the shared design principles and assumptions of these two approaches.

4.2.3 Design assumptions

Assuming the VM is already launched, we may model the hypervisor intervention as two transition functions ϕ and γ that update the hypervisor state and output, respectively.

$$\phi : VM_{state} \times S_{hyp} \rightarrow S_{hyp} \tag{4.1}$$

$$\gamma : VM_{state} \times S_{hyp} \rightarrow VM_{state} \tag{4.2}$$

Both ϕ and γ take as external input the VM_{state} which corresponds to the CPU registers, VM_{cpu}, and the VM memory, VM_{mem} at the time of a VM exit. In order to reflect only the effects caused by untrusted guests, we simplify our equations to include only controllable variables of the guests, i.e., VM_{state}, excluding non-controllable events, e.g., host asynchronous interrupts during the hypervisor intervention. The ***hypervisor behavior*** is a sequence of hypervisor interventions ($VM\_exit\_trace = \{VMexit_1, ..., VMexit_N\}$), representing the flow of *VM exits* triggered by a workload to reach a valid VM state;

By these two equations, we derive the following considerations.

- First, the resulting VM_{state} in function gamma of Equation 4.2 may be adopted to evaluate the accuracy of the replay: if the replay is not accurate, the state of the hypervisor will not be updated properly, and, as an indirect consequence, also the resulting output. So, we collect some VM_{state} updates, observed after an hypervisor intervention, to compare the replay execution with the nominal execution.

- Second, the S_{hyp} mostly depends on the VM_{state}, so, when we revert the snapshot of a guest VM, in some way, we are also updating part of the hypervisor state that reflects the VM state. Therefore, instead of reverting the entire hypervisor state (infeasible when nested virtualization is not enabled), we may accept the VM state revert as sufficient to revert also the hypervisor state.

- Third, the target events to record during the hypervisor intervention (***VM seed***) are the accesses to the VM state VM_{state}, as, by replaying the hypervisor intervention with the right VM state as external input, we should observe the same VM state updates.

In our design, we deliberately avoid recording the VM_{mem} as the only part of the VM state that we do not monitor. This choice was made to reduce the complexity of VM seeds. In the next section, we demonstrate that this choice is a good compromise to accurately record hypervisor behaviors, and we discuss the few cases in which IRIS can not obtain good accuracy. Therefore, the ***VM seed*** includes the pairs of VMCS {field, value} read via `VMREAD` instructions, and the values of general-purpose registers (GPR), both obtained during the handling of a VM exit within the *VM_exit_trace*.

Finally, we define *accuracy* of the replaying mechanism as its ability to reproduce a valid hypervisor behavior for the recorded metrics, i.e., code coverage at the hypervisor level and writes performed into the VMCS fields or GPR registers. Code coverage at the hypervisor level is the simplest metric to estimate how much replaying *VM seeds* is *accurate* compared to the recorded hypervisor behavior. However, coverage-related metrics are expensive in terms of latency and manual effort when not supported by hardware features. To mitigate this point, we also provide new synthesized metrics that are peculiar to the hardware-assisted hypervisor solutions. Specifically, in the case of on-device solution, we used the code coverage

as accuracy metric, as the target is the hypervisor running on the device, instead, in case of the transparent replay, we only compare the `VMREAD`s, `VMWRITE`s obtained during the VM exit handling and the GPR obtained in output of a VM exit handling.

Record

The recording component captures information during the VM's execution. When a VM exit occurs, triggering hypervisor intervention, a trace is created for each exit in the form of a VM_exit_trace, categorized by an exit reason. The recording architecture logs: Accesses to the VM states, storing them in a VM seed structure. Specifically, it captures accesses (i.e., VMREADS) to the VMCS guest-state area and the general-purpose registers (GPRs) prior to hypervisor intervention. Optionally, it can record metrics, such as updates to the VM states (i.e., VMWRITEs to the VMCS), GPR registers after hypervisor intervention, and the CPU cycles consumed during the VM exit. In the case of IRIS, we also instrument the target hypervisor (Xen) to provide VM exit code coverage after the intervention.

Replay

To replay only hypervisor interventions, we need tp continuously triggering VM exits and directly activating the VM exit handler without executing the guest OS. To this end, the replay component requires two key functionalities: the controlled generation of VM exits from the guest OS to the hypervisor target, and the submission of recorded VM seeds during these exits. When a VM seed is submitted, the hypervisor will process the corresponding VM exit handler based on the exit reason.

To this aim, we use a dummy VM into which the VM exits are injected. The dummy VM is initialized to the same state as when the recording was activated. This state could be restored from a snapshot of the VM or any other stable point, such as prior to the VM boot sequence, where no instructions have yet been executed. This approach ensures that the record and replay phases start from an equivalent VM state with the corresponding hypervisor data structures.

In cases of negative testing, the framework supports submitting recorded

VM seeds to update the hypervisor state, and crafted VM seeds to test hypervisor interventions under corner-case conditions.

Chapter 5

Implementation

We provide technical details about the tool implementations of this thesis. We first introduce CloudPerf in Section 5.1, which is the tool adopted to evaluate memory access isolation and Intel MBA. Afterwards, we introduce IRIS in Section 5.2, which is the reference on-device implementation of the hypervisor behavior record and replay, and HyRo in Section 5.3, which is a work-in-progress implementation of the transparent replay.

5.1 CloudPerf implementation

CloudPerf includes a Linux kernel module and user-space applications. We implemented it on Linux to simplify deployment and make it easier for the open-source community to extend.

Kernel module generic configurations. The kernel module includes some macros that describe the *target architecture*:

1. CAS: it configures and samples events on each available memory channel (architecture-specific for Xeon scalable processor)
2. MBA: architecture specific for new Xeon Scalable processors
3. NO_SMT: available on all recent x86 architecture when hardware threads (simultaneous multithreading) are disabled
4. PMC_CORE: available on all recent x86 architecture

The kernel module includes some macros which describe the *operating mode* of the tool:

- **Periodic sampling:** The tool is configured to periodically sample a workload running in userspace. The user may choose to start the periodic sample once the kernel module is inserted[1], or by userspace interfacing with the kernel driver[2].
- **Latency sampling:** the tool is configured to sample the latency of a workload. The workload may run in userspace [3], or in kernel space. In kernel space, the user can decide to start the workload at the time of inserting the module[4], or by interfacing with the kernel module by userspace[5].

Running the workload in kernel space allows the user to run privileged instructions without making a context switch, e.g., to access the performance counters and disable/enable the kernel preemption. The user may choose a periodic or latency sampling based on the kind and goals of the analysis. The latency sampling allows the user to analyze more in detail the workload execution because the user has more control over the environment parameters and the user may limit the workload to a few instructions. The latency mode is the mode that we adopted in our experiments to evaluate the indirect limitation of MBA.

Periodic sampling Through our tool, we can periodically sample events on a specific set of online cores. We allocate a data structure on each online CPU to store the samples. After finishing the sample, we save the sampled data on files (one file for each core). We can start the periodic sampling in two ways: by inserting the kernel module (insmod cmd) or via the system-call read from the device driver (driven by usr space).

Let's consider an example of periodic sampling. First, the user compiles the kernel module defining the macro "MOD_HR_TIMER" to spe-

[1]MOD_HR_TIMER_ON_INIT: periodic sample started once the kernel module is inserted

[2]MOD_HR_TIMER: periodic sample started by userspace

[3]MOD_SYNC_SAMPLE: latency sampling with the workload running in userspace

[4]MOD_LATENCY_BENCH_KERN_ON_INIT: latency sampling with the workload running in kernel-space and started when inserting the kernel module

[5]MOD_LATENCY_BENCH_KERN: latency sampling with the workload running in kernel-space and started by userspace

cialize the module to make a periodic sample and to define the variables shown in list. 5.1. List. 5.1 shows the kernel module configurations in case the user desires to sample each millisecond of the performance events on top of eight cores, i.e., one core where the kernel module is running plus seven "interference" cores (INTERFERENCE_CORES 7). The module is configured to work as a device driver (CH_DEV), so the user may send commands from userspace by interfacing with the character device. Moreover, the user setups the module to activate the MBA between the 8 and 13 periods ("PERIOD_START_MBA" and "PERIOD_STOP_MBA"). At period 8, on every core, the user may choose to change the delay of a class of service affecting all cores associated with that class (e.g., $delay90 \rightarrow CLOS1$), or change the CLOS associated to the current core (e.g., change $CLOS0 \rightarrow CLOS1$ on the current core). For instance, in our case, the user maps different delays to $CLOS_1, CLOS_2, ..., CLOS_7$, and, at the 8 periods, every interference core i changes its class of service from $CLOS_0$ to the corresponding $CLOS_i$. To specify these parameters, the user defines them as kernel parameters when inserting the module. In userspace, the user runs multiple memory-bound threads, one for each core. Once synchronized, the user starts the periodic sampling by reading "/dev/sampling" from the device driver.

Note that we support some additional scripts to

1. define the workload on core zero and the stress workload of the interference cores
2. define the range of delays to be tested on the main core and on the interference cores
3. the number of observations
4. compile the kernel module, the workload, and the main program

Listing 5.1. Periodic sampling parameters

```
#ifdef MOD_HR_TIMER
    #define NSAMPLES 100
    #define HR_TIMER
    #define CH_DEV
    #define PERIOD_US 1000
```

```
    #define COARSE_GRAINED_EVENTS
    #define WR_FILE_EXIT
    #define WR_FILE_EXIT_DIFF 1
    #define NAMEFO
    "./OutputPeriodicSamplePeriodSampleDiff-cpu.csv"
    #ifdef MBA
        #define PERIOD_START_MBA 8
        #define PERIOD_STOP_MBA 13
        #define MBA_TRANS
        #define CHANGE_COS
    #endif
    #define INTERF_CORES 7
#endif
```

Latency sampling. Intel processors support a time-stamp counter (TSC) to capture a time-stamp at a fixed frequency. However, this instruction is subject to instruction reordering (i.e., Out-of-Order Execution) by modern processing elements, which can introduce noise into the measurements. In the worst case, the instruction may be delayed until all other instructions stored in the reorder buffer (ROB) have been executed. Modern reorder buffers may store more than two hundred instructions, so the variability in the measurement may not be negligible, as the timestamp may be delayed during the workload execution.

To maintain sequential order for the timestamp, we use serialization instructions (e.g., fence, mfence, and lfence), which enforce the commitment of instructions (or specific instructions) before proceeding with the next programmed instruction. We divide the process into five distinct phases separated by serialization instructions.

The first phase includes all the steps necessary to configure the environment and experiment parameters. For example, if we want access to a specific memory area to cause last-level cache misses, we may set up this phase to flush that memory area from the cache before running the actual workload. Additionally, the user may choose to disable OS preemption to ensure that the workload is not preempted in favour of another task or to enable technologies, e.g., the Intel MBA to apply regulation.

The second phase reads the performance event counters to establish a baseline before the workload execution. Note that the counters are

saved in local variables, which correspond to CPU registers after program compilation.

The third phase reads the timestamp (*start_timestamp*) before running the serialization instruction (in our case, lfence), to estimate the execution time of the serialization instruction. After the serialization instruction, we are certain that the first instruction to execute is the programmed instruction, so by reading the timestamp again (*middle_timestamp*), we can obtain the execution time of the serialization instruction by calculating the difference between the middle and start timestamps. Afterwards, we run the workload.

To conclude the evaluation, we read the timestamp (*end_timestamp*) and the performance events from the counters. This operation is performed after running another serialization instruction that ensures the workload completes every single instruction before actually reading the performance events. This final serialization instruction needs to be subtracted from the total execution time of the workload. Consequently, the execution time of the workload is equal to the workload execution time (*end_time* - *middle_time*) minus the execution time of the serialization instruction (*middle_time* - *start_time*).

As a final step, we write the performance events, obtained as the difference between the initial and final counts, and the workload execution time to a file stored on disk. Before returning, we provide the user with the ability to revert the environment settings to normal, such as re-enabling preemption.

Fence instructions and TSC reading can be executed in userspace, while access to performance counters requires the privileged mode of kernel space. Below, we compare two different measure implementations of latency sampling. The "userspace implementation" runs the workload in userspace, requiring the context switch for the event monitoring. To avoid preemption, the userspace task changes the scheduler to SCHED_FIFO and sets its priority to a high priority. In the second implementation (kernel space), instead, we execute the workload in kernel space. In kernel space, we can stop the preemption by disabling hardware interrupts. However, the workload in kernel space has limitations in the size of dynamic memory to allocate. In tab. 5.1, we briefly list the main differences between these two implementations.

Table 5.1. Timing Differences in User space/kernel space implementation

Metric	User-space	Kernel-space
event measure	overhead (switch to kernel space)	minimum overhead
latency measure	minimum overhead	minimum overhead
no preemption	FIFO policy	disable hw interrupts
Workload size	no limit	around 4 MB

To show the differences between these two ways of sampling workload latencies, we provide the following example. In this example, the target workload to measure reads 256 KB of memory, which corresponds to 8192 LLC misses. In Figure 5.1, the workload is executed in userspace and in kernel space, 60 times for each. The boxplot in Figure 5.1, summarizes the statistical features for "the total number of LLC miss" for the two cases. In kernel space, the total number of LLC misses observed during the workload execution almost is 8192 except for two outliers. So the experiments are reproducible. Contrarily, the number of LLC misses in user space does not include 8192 in the 25 -75 percentile range. Furthermore, there are more outliers in userspace. Probably, the system call invoked to read the performance events causes the increasing number of LLC misses. To confirm this hypothesis, we need to prove that the additional LLC misses are confined to kernel space. Therefore, we specialize the number of LLC misses in user and kernel space for the userspace implementation.
The Figure 5.2 shows the new box plot for the userspace, considering only the LLC misses in userspace. This time, the value 8192 is included in the 25th percentile.

In conclusion, when possible, we do our evaluations in kernel space. However, if we need larger workloads, we run the workload in userspace.

Performance events. Our tool allows users to specify the performance events to monitor during workload execution. The tool supports eight or four per-core events, depending on whether hyper-threading is enabled (the number of per-core events is limited by the architecture). Table 5.2 shows an example of configured events to trace. These events can be configured to trace in userspace, in kernel space, or in both spaces.

Actually, our tool also supports two uncore events from the Integrated Memory Controller (IMC) box of the Intel Xeon Processor. The uncore region of the microarchitecture includes additional performance counters, which the tool may include in future updates. The per-core PMCs and

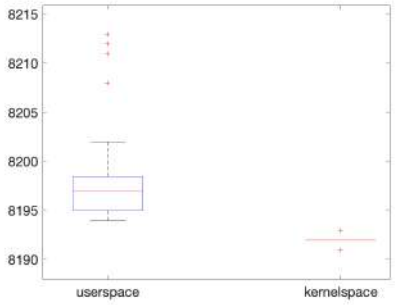


Figure 5.1. Box plot of the total LLC misses for a 256 KB memory load (8192 requests) for user and kernel space implementation

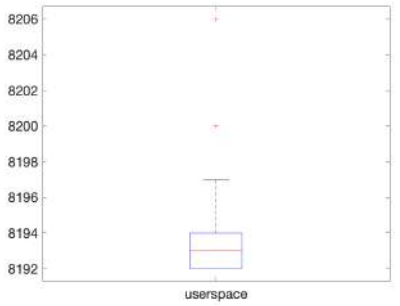


Figure 5.2. Userspace implementation concerning only userspace LLC misses

some of the uncore PMCs are controlled by the tool via Model-Specific Registers (MSR), while the uncore performance counters of the IMC box are accessed via the PCI Express interface. To manage the uncore performance counters, our tool follows the procedures illustrated in the reference manual [65]. Initially, it invokes a *setup* function to map the Intel PCI physical addresses to the virtual space of the tool's kernel module, and then freezes, resets, and configures each PMC. After this setup, the tool can invoke *start* and *stop* functions to begin and end counting, respectively. Listing 5.2 provides a high-level description of the driver for these performance counters. "Freezing the performance counter" refers to stopping its count, while "resetting the PMC" means setting its count back to zero. We ensure that the performance counter is frozen before any configuration is applied.

Listing 5.2. Uncore pmc driver

```
Setup:
1) memory mapping
2) freeze and reset each pmc
3) configure the pmc

Start:
1) unfreeze the pmc
```

```
PMC

Stop:
1) freeze the pmc and reset
2) memory unmapping
```

Table 5.2. Example of Percore PMC events

PMC	Event	Space
pmc1	retired memory load instruction	OS
pmc2	retired memory laod instruction	USR
pmc3	retired memory load instruction with LLC miss	OS
pmc4	retired memory load instruction with LLC miss	USR
pmc5	llc references	USR
pmc6	llc misses	USR
pmc7	retired instruction	OS
pmc8	retired instruction	USR

5.2 IRIS implementation

The current version of IRIS is built upon the Xen hypervisor. We chose Xen since it is an open-source hypervisor and supports *Hardware Virtual Machine* (*HVM*) mode. HVM mode tries to make full virtualization easier, using the hardware emulation to accelerate CPU virtualization (privileged instructions) and the MMU (page tables). Figure 5.3 depicts the high level architecture of IRIS. IRIS is implemented as a set of patches for the Xen kernel. All IRIS components code is written in C language. The details of each component are briefly explained in the following.

Record. Currently, in IRIS we obtain code coverage at the hypervisor level via compile-time instrumentation approaches using *gcov* [131]. The hypervisor codebase should not be instrumented as a whole since we need to avoid most sources of non-determinism, e.g., due to interrupts, kernel threads, and statefulness. We selectively instrument hypervisor components crucial for VM exit handling, such as the abstraction of vCPU, HVM domain-specific functions, and the handler of VMX-related operations. While running, the resulting instrumented binary will write its own

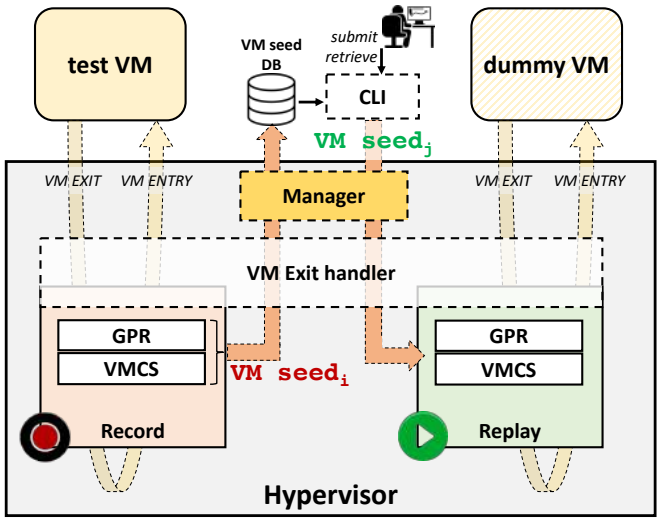


Figure 5.3. Overview of IRIS design

basic block coverage to a bitmap, which is exported as a shared memory area accessible at the guest level. We remark that code coverage is cleaned up by removing hits due to the execution of our record and replay components. Further, code coverage information can be retrieved for each VM seed submitted. Regarding the reads and writes performed into the VMCS fields, the hypervisor uses machine instructions `VMREAD` and `VMWRITE`, wrapped respectively by Xen `_vmread()` and `_vmwrite()` functions. We instrument these functions by adding a callback function invocation to store pairs of VMCS {field, value} read or written in the shared memory area. Regarding the values of guest GPR, they are stored in Xen data structures during VM exit handling, since they are not included in the VMCS. The current implementation, for each VM seed, uses an array of structs to store GPR, VMCS fields read or write. The struct is defined to store: i) a flag (1 byte) that indicates the kind of data; ii) the encoding (1 byte) of GPR (15 values) or VMCS fields (147 values); iii) the value (8 bytes) for GPR or VMCS field. Once again, we invoke a callback function at the start of the VM exit handler execution, to also buffering this kind of data. The temporal metric can be retrieved using instructions to get CPU-cycle counters. For example, Intel provides the

`RDTSC` instruction, which reads the current value of the CPU's time-stamp counter.

Replay. The replaying component enables the continuous triggering of VM exits by leveraging a *dummy VM* running in HVM mode. A possible solution to continuously submit VM exits is to let the dummy VM do a first exit and then implement a loop directly within the *VM exit handler*. However, a loop in *root mode* could be detected from the hypervisor as a hang, leading to forced crashes. In addition, such a direct loop avoids the VM entry (see section 3.2), which occurs as the last step in the VM exit handling. The *VM entry* operation includes several checks on the VMCS fields (specified in Section 26.3 in [66]) that are representative or real VM behavior and are used to guarantee semantically-correct VM seeds submission. Thus, our replaying architecture lets the VM exit handler execute the VM entry and then forces the *dummy VM* to immediately trigger another VM exit, preventing the execution of any instruction at the guest level.

We implemented the VM exit/entry loop by leveraging an ad-hoc hardware feature of Intel VT-x, the Intel *VMX-preemption timer*, for the *dummy VM*. The *VMX-preemption timer* counts down (from the value loaded by a VM entry) in VMX non-root operation, at a rate proportional to that of the timestamp counter (`TSC`). When the timer counts down to zero, it stops counting down and a VM exit occurs (see Sections 25.2, 25.5.1, 26.6.4 [66]). In our framework, a preemption timer value set equal to zero allows the hypervisor to preempt the *dummy VM* execution before the CPU executes any instructions in the guest. Regarding *VM seed* submission, we implement callback functions inserted at compile-time and invoked during the VM exit handling to submit GPR and VMCS values according to the *VM seed* values. The GPR values are simply copied to the corresponding hypervisor data structures. The VMCS values can be written into the VMCS by invoking the `_vmwrite()` function. However, this solution is adopted only for writable VMCS fields, since some of them are read-only. For the latter scenario, we instrument the function `_vmread()` by inserting a callback function to replace the values returned from the readings on the VMCS with those submitted with the *VM seed*.

Manager We implemented a component called IRIS *manager*, which exposes an interface that can be used by a *user-space application (CLI)* to

(i) choosing between operation modes, i.e., *record* and *replay*; *(ii)* retrieve *VM seeds* and metrics during the *record mode*; *iii)* submitting *VM seeds* during the *replay mode*. When the IRIS *manager* enables the *record mode*, it runs a *test VM* and allows it to trigger normal VM exits as they occur. The *record mode* can be configured to store *VM seeds*, metrics, or both of them. Recording can be manually or programmatically stopped after a given number of monitored VM exits. After a specific time of recording, the IRIS *manager* allows keeping the *test VM* in an idle loop, reading for a new recording session; otherwise, the *test VM* continues execution with no recording enabled. In the *replay mode*, IRIS *manager* first runs a *dummy VM* (optionally by reverting to a particular VM state) and then allows users to submit seeds on-demand. Also, in this case, the *manager* puts the *dummy VM* in an idle loop to wait for new *VM seeds* to submit. When a new *VM seed* is available for submission, the replaying component submits it to the hypervisor. Note that both recorded seeds and manually crafted seeds can be submitted at this step. Finally, the IRIS *manager* allows enabling the *replay mode* together with the *record mode* enabled to store metrics while replaying. This latter is necessary to evaluate the accuracy and efficiency of recorded/crafted *VM seeds* which are submitted via the *replay mode*. The manager component consists of a backend driver at the hypervisor level. The interface exposed to users is implemented using the *hypercall* mechanism. We implemented the `xc_vmcs_fuzzing()` hypercall to trap into the hypervisor, to enable and control the recording and replaying phases. Given that, a user-space application (IRIS CLI) invokes such hypercall to enable the functionalities provided by IRIS manager. Finally, the manager uses `copy_to_guest()` and `copy_from_guest()` Xen routines to respectively retrieve recorded VM seeds and metrics and submit *VM seeds*.

5.3 HyRo implementation

The current implementation of the HyRo framework is built on top of KVM, using Linux Kernel version 5.19.17, deployed on Ubuntu 22.04.03 LTS. The reference CPU architecture is Intel x86, leveraging hardware-assisted virtualization extensions. It requires only basic features, such as EPT and VMCS, and all components are implemented in C.

In order to enable the record and replay features in KVM, in kernel space, we mainly modified the KVM modules related to x86 virtualization (vmx.c) and nested virtualization (nested_vmx.x) and we added two new kernel modules (HyRo.ko and fuzzer.ko), which act as a middleware layer between user space and kernel space. Instead, the orchestrator and the mutating core are distributed as user-space applications.

Record. For each VM exit of the guest VM, the framework records the (1) input GPRs, i.e., the general purpose registers of the guest before the hypervisor handles the VM exit, (2) the read/write on the specification (VMCS) of the guest VM, and (3) the output GPRs, i.e., the general purpose registers of the guest after the hypervisor intervention. Figure 5.4 shows the steps of recording in HyRo. In nested virtualization, the VM

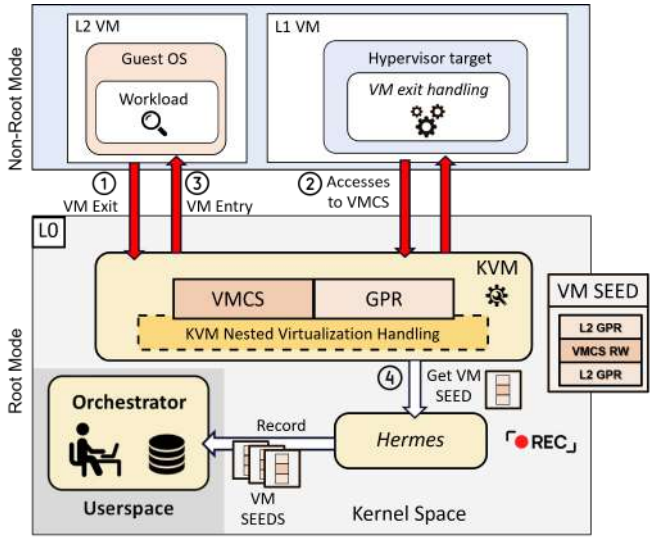


Figure 5.4. VM exit recording

exits caused by the guest VM L2 (1) are handled by the L1 hypervisor or by the hypervisor host L0. If the L1 hypervisor is selected to handle the VM exit ("the VM exit is reflected"), the `nested_vmx_reflect_vmexit` calls the `nested_vmx_vmexit` function in order to prepare "the reflection" of the VM exit, which includes the setup of the VMCS12, i.e., the specification of the guest OS. At this stage, HyRo records (i.e., stores) the

L2 General-Purpose Registers (GPRs), referred to as the "input GPRs." Afterwards, L0 resumes the execution of L1 to handle the VM exit of L2. The L1 hypervisor performs a series of operations on VMCS12 ②, including instructions such as VMREAD and VMWRITE. In order to trap the VMREAD and VMWRITE instructions, we disable the VMCS shadowing, as, in this condition, KVM performs the emulation of the VMREAD and VMWRITE operations. During the emulation of VMREAD and VMWRITE (functions `handle_vmread` and `handle_vmwrite`), HyRo makes the trace of these operations, capturing both the field accessed and the value read or written in the VMCS by the L1 hypervisor. At the end of handling the VM exit of L2, L1 performs the resume of L2 ③, which is trapped and emulated by L0. At this point, HyRo makes the recording of the L2 GPRs, referred to as "output GPRs" because they pertain to the VM entry process.

In conclusion, for each VM seed, the framework creates an array of structures to store the input/output General-Purpose Registers (GPRs) and the VMCS (Virtual Machine Control Structure) fields that are read or written. The structure is defined as follows: (i) a start register of 13 bytes, serving as a delimiter; (ii) VM Exit registers, occupying 195 bytes; (iii) VMCS operations (which typically vary in number, but may take up to 320 bytes); (iv) an operation counter of 1 byte; and (v) VM Entry registers, covering 195 bytes. Crafting and sizing VM seeds is a challenging task, particularly within the mutation process. Therefore, this structure was designed to minimize seed complexity and mutation surface as much as possible. By using shared memory between the HyRo.ko and kvm-intel.ko Linux kernel modules, the framework can extract this data during each VM exit without interrupting KVM's control flow. This data extraction is seamlessly managed, following instructions from the orchestrator interface, ensuring KVM continues to operate without disruption.

Replay. The primary objective of the replay component is to control the generation of VM exits from the guest OS L2 to the hypervisor target L1. To this aim, we extended the current nested virtualization implementation of KVM, especially the `nested_vmx_run` function, to aid our replay framework the capability of injecting custom VMEXIT events based on the instructions provided by the orchestrator. Figure 5.5 shows the step of replay in HyRo. Before of calling the intervention of the hypervisor target,

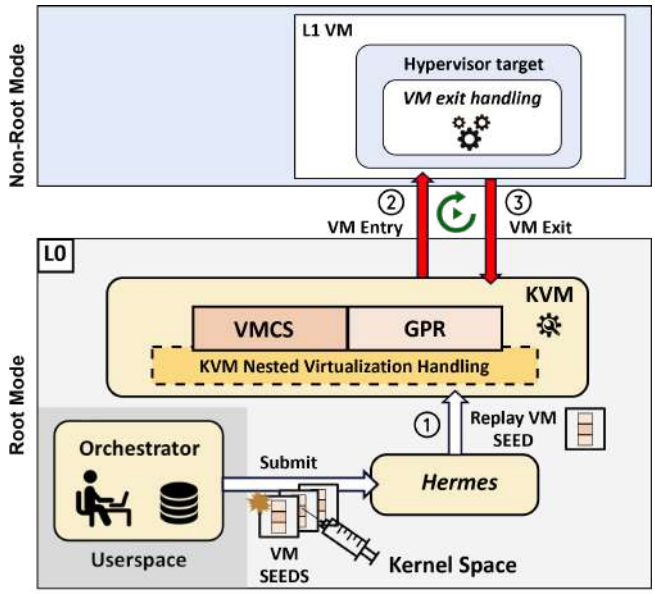


Figure 5.5. VM exit recording (a) and VM exit submission (b)

the orchestrator is powered with the ability to define the specification of the guest OS (VMCS12) and the general purpose registers of the guest OS, i.e., the VM seed to submit, in order to specialize the intended VM exit ①. Define the VM seed to submit, our implementation (i) verifies the active vCPU specification is the one corresponding to the target L1 hypervisor, (ii) loads the VMCS12 host state area, and, finally, resumes the hypervisor target ②. To regain the control after the L1 intervention, the orchestrator intercepts the VMRESUME instructions ③ issued by L1, as, if the VM exit handling is successful, L1 will resume L2 by running this instruction.

Due to its nature, the replay component tends to be more invasive than the recording component, which can make it prone to potential performance bottlenecks, including resource starvation. To address this issue and prevent a vCPU, or in some cases a physical core, from becoming unresponsive, HyRo incorporates timeout mechanisms. These mechanisms are designed to relinquish control back to KVM if HyRo detects prolonged hangs or delays, ensuring that the system remains responsive and avoids

deadlocks.

It's also worth noting that the recording component can be activated during the replay phase. This enables the collection of information regarding VMCS coverage, which can provide valuable insights for monitoring and analysis during the replay process

Orchestrator. HyRo includes a user-space interface for managing the recording and replay APIs, which are accessed and abstracted by the orchestrator component.

During the recording phase, the user may record events occurring within the L2 guest VM. Specifically, the orchestrator allows the user to specify the beginning of recording, the number of VM exits to be recorded, and if enabling the VMCS dumping during the hypervisor intervention. Once the recording phase is completed, the orchestrator generates a raw seed file, designed to reduce the size of the recorded information and containing encoded information collected during the VM exits. At this point, the user can request the orchestrator to present the raw seeds in a human-readable format. It is worth noting that the orchestrator also permits the termination of the recording phase at any point, even if it has not reached the originally defined parameters, such as the predetermined number of VM exits to record.

On the other hand, the orchestrator provides the option to initiate a replay phase, during which the user can decide how many VM exits to submit and from which specific exit to begin the replay. Before starting the replay, the user may enable the recording of VM_{state} coverage metrics, used to estimate the replay accuracy.

The orchestrator provides users with the flexibility to decide whether to include mutations when submitting a seed using a specifically developed fault injector.

The orchestrator is a userspace application that communicates with the Linux module through ioctl commands. The HyRo.*ko* Linux module utilizes shared variables with the "*kvm.ko*" and "*kvm-intel.ko*" modules to keep track of various commands, such as enabling replay or recording. These shared variables serve as a means of communication and coordination between the modules, allowing them to provide instructions to each other based on the values of these variables. This approach enables effective and dynamic control of the virtualization environment, allowing

for the activation and deactivation of specific features or functionalities as needed.

The interface provided by "HyRo.*ko*" allows the orchestrator to perform various actions on the buffer that represents the recorded or replayed VM Exit data. These actions include reading and writing on the VM_{state}, as well as retrieving VM_{state} coverage metrics.

Chapter 6

Evaluation

This Chapter includes the research questions, the experiments and the findings of the thesis. Section 6.1 responds to the research questions regarding memory access isolation, while Section 6.2 evaluates the research questions regarding IRIS, the reference on-device implementation of the hypervisor record and replay.

6.1 Memory Access Isolation

- **RQ1. MBA Throttling:** How much is the memory bandwidth of physical cores using Intel's MBA technology?
- **RQ2. MBA Guarantees:** Can we provide memory bandwidth guarantees to a critical core using Intel's MBA technology?
- **RQ3. MQO as Interference Detector:** Can the variation of the memory queue occupancy reflect the number of cores co-accessing the memory?

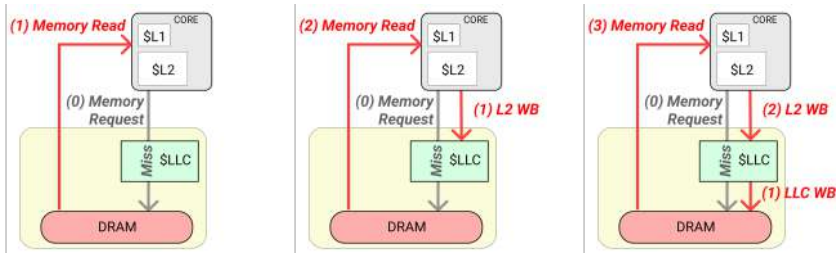
6.1.1 Methodology

We use only one socket and set the number of active cores via BIOS configuration. We used only one memory channel in both analyses. We set the DRAM refresh rate to periodic and the CPU core frequency to maximum (no power saving) in the BIOS. These parameters help keep

the minimum latency variability and maximize the memory request rate. Similarly, we disabled the prefetcher and hyper-threading to reduce variability since hardware threads on the same physical core share the same "queue" to the memory and the prefetcher can preload several lines after a single cache miss. Experiments are conducted on an Intel Xeon Silver processor (Cascade-Lake micro-architecture) with eight cores. Each core runs at 2.1 GHz and has a 512 KB L1 cache (256 KB out of 512 KB are dedicated to instructions), and a 1MB L2 cache. All cores share an 11MB LLC.

We use three different synthetic memory workloads to trigger different traffic patterns of the microarchitecture which go through the MBA and target the memory subsystem. The memory read workloads, i.e., RI and RII, consider only synchronous memory accesses, while the memory write workload also accounts for the asynchronous memory writes. Contrary to the existing works [124, 143], we designed a new memory workload, i.e., RI: Exclusive DRAM Bomb, capable of triggering memory reads without causing L2 WBs. Figure 6.1 shows the generated events caused by one memory request of each workload. We chose 8192 requests (meaning 512 KB memory fetches because each request retrieves a cache line size of 64 bytes) since it assures us to not have L2 WBs in R^I (L2 WBs = 0), given the L2 cache size of 1MB.

In the experiments, our measures achieve the 99% accuracy calculated as the 99% mean confidence interval using a student's t distribution divided by the sample mean.



(a) RI **(b)** RII **(c)** WII

Figure 6.1. Synthetic workloads

MBA delay values. Intel provides nine possible delay values between 10 to 90 with a stride of ten. These values have a linear effect in delaying the requests going to the LLC. No additional information is provided from Intel. We exclude 20 and 30 delays from the analysis as they are erroneous on our processor model. As per the errata [62] provided by Intel, these two delay values produce the same results as the 10 delay value.

RI: Exclusive DRAM Bomb. The RI workload is composed of 8192 read requests where each request causes one LLC miss ($readReqs = LLCmisses = CASReads = 8192$). The CAS R is the command issued on the DRAM channel. Instead of RI, the RII workload, represented in Figure 6.1 (a), is capable of not causing one L2 WB for each memory request. We call this workload "DRAM Exclusive Bomb", as this behaviour can be reproduced in exclusive microarchitectures. To reproduce this workload, we made the following operations before its execution: we allocate, read and flush a memory address space as big as the L2 cache, (i) we flush the memory space of the workload (8 KB of dynamic memory) to read. This way, the workload's memory requests to the workload's memory space will cause memory reads, as we flushed the space, without L2 WBs, as the private cache is empty.

RII: DRAM Bomb Read. The RII workload is composed of 8192 read requests where each request causes one LLC miss and one L2 WB ($readReqs = \text{LLC}misses = CASReads = L2Wbs = 8192$), represented in Figure 6.1 (a). Sohal et al. [124] adopted this workload to evaluate the regulation of the MBA. This workload is mostly adopted to generate memory interference. To reproduce this workload, we made the following operations before its execution: (i) we flush the memory space of the workload (8 KB of dynamic memory) to read, and (ii) we read memory enough to fill the private L2 cache space. This way, the workload's memory requests to the workload's memory space will cause memory reads, as we flushed the space, and L2 WBs, as the private cache is full.

WII: DRAM Bomb Write. We have memory writes in a WB policy when a dirty LLC cache line is evicted. In order to reproduce the worst-case scenario, WII for each memory request causes one memory read, and one memory write operation due to the cache line eviction and substitution. Figure 6.1 (c) summarizes the events generated from each memory request of this workload. The W^{II} in Table 6.1 represents this

MBA delay $d \in D$	
D	$10; 40; 50; 60; 70; 80; 90$
Workloads $k \in K$	
K	$R^I : \{8192CASR, noL2WBs, noLLCWBs\}$
	$R^{II} : \{8192CASR, withL2WBs, noLLCWBs\}$
	$R^{III} : \{8192CASR, withL2WBs, withLLCWBs\}$

Table 6.1. MBA delays and designed workloads

kind of workload. In particular, the W^{II} workload is composed of 8192 requests where each request is the cause of one memory read, and one memory write ($writeReqs = LLCmisses = CASReads = CASWrites = 8192$). Only the second operational condition (II) is possible for the write workload since an LLC WB can only result from an L2 WB.

6.1.2 MBA Throttling

In order to measure the indirect memory access limitation for an interfering core when MBA is enabled with different delays, we measure the execution latency ($L_{k;d}$, $k \in K, d \in D$) of the three workloads ($k \in K, K = \{R^I; R^{II}; W^{II}\}$) running on top of one physical core under MBA throttling $d \in D, D = \{10; 40; 50; 60; 70; 80; 90\}$.

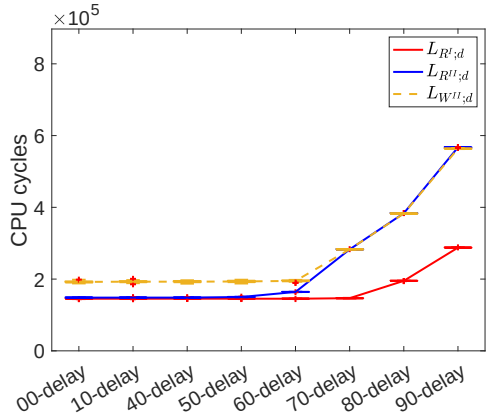


Figure 6.2. Latencies for the different workloads and interpolation of the respective mean latencies

Figure 6.2 shows the $L_{k;d}$ boxplots and a mean interpolation for the observations of the same workload and different MBA delays. Observing the boxplot width, the latency variability is the same (or low) when the limitation is enabled or disabled (0-delay).

MBA Variability

Finding 1: The controller MBA is not a new source of variability. The main source of variability of memory requests is the same with and without MBA, e.g., coming from the periodic DRAM refresh cycles.

Figure 6.2 shows two effects clearly. (i) The limitations do not intensely affect the relative 0-delay observations until the 70-delay in our setting. (ii) The boxplots of workload R^{II} and W^{II} overlap for high delays, meaning a similar behavior due to L2 WBs.

To better interpret the results from Figure 6.2 for the 70, 80, and 90 delays, we prefer to go from the latency to the bandwidth domain. If T is the latency $min(L_{R^I;0})$ to synchronously fetch 8192 cache lines ($N = 8192$) without throttling[1], we calculate the corresponding number of fetched cache lines in T with throttling (memory read operations in T) of a generic configuration $k; d$, where k is the workload, and d is the delay, as[2]:

$$Req_{k;d}^{sync} = \frac{T * N}{L_{k;d}};$$

We report in Table 6.2 the maximum observed bandwidth for delays from 60 to 90. Hence, the resulting percentage of synchronous memory bandwidth with regulation is calculated as the number of memory reads fulfilled in T instead of N:

$$BW_{k;d}^{sync}\% = \frac{\frac{Req_{k;d}^{sync}}{T}}{\frac{N}{T}} * 100 = \frac{Req_{k;d}^{sync}}{N} * 100$$

These results are summarized in Table 6.2. We are interested in showing

[1] 8192 is our chosen reference of memory requests, as explained earlier

[2] Req is the number of memory reads completed in T with throttling instead of N ($BW_{k;d} : L_{k;d} = N : T$)

these percentages since in our work we intend to allocate portions of the synchronous memory bandwidth to each physical core.

Note that the synchronous memory bandwidth corresponds to the reached memory bandwidth for RI and RII ($BW^{sync}_{k;d}\% = BW_{k;d}\%, k = R^I, R^{II}$), as no asynchronous memory requests are caused by these workloads. Instead, the reached memory bandwidth of WII is two times the synchronous memory bandwidth ($BW_{W^{II};d}\% = 2 * BW^{sync}_{W^{II};d}\%$), as, for each synchronous request, WII causes one memory write.

Table 6.2. The completed synchronous operations (Req) in T when the workload is subject to regulation and the resulting available percentage of synchronous memory bandwidth.

Metrics	60-delay	70-delay	80-delay	90-delay
$Req^{sync}_{R^I;d}$	8192	8079	6080	4105
$BW^{sync}_{R^I;d}$	100 %	98%	74%	50%
$Req^{sync}_{R^{II};d}$	7237	4183	3080	2084
$BW^{sync}_{R^{II};d}$	88%	51%	38%	25%
$Req^{sync}_{W^{II};d}$	6376	4183	3091	2097
$BW^{sync}_{W^{II};d}$	77%	51%	38%	26%

The major results from this analysis are the following.

Exclusive DRAM Bomb implications. The less restrictive limitation of MBA 1.0 for a memory read workload is when there are no L2 WBs.

$$BW_{R^I;d} > BW_{R^{II};d} \tag{6.1}$$

$$BW_{R^I;70:90} = 2 * BW_{R^{II};70:90} \tag{6.2}$$

We highlight this behavior in Figure 6.4 where, once flushed the private L2 cache, we fetch 2048 KB memory in blocks of 64KB under 90-delay regulation. The fetching of the first 11 blocks fills the private L2 cache without causing L2 WBs, i.e., RI. Afterwards, the occupancy of the private cache grows causing L2 writebacks to the LLC, i.e., RII. The 90-delay regulation observed for the RII is half of the RI, confirming what is specified in Table 6.2.

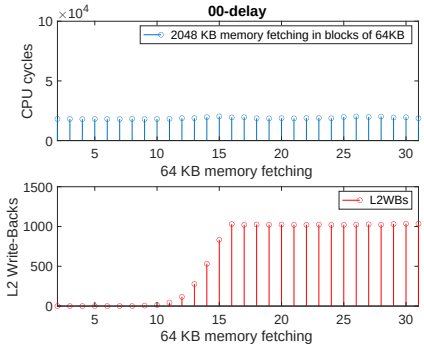


Figure 6.3. 2048 KB memory fetching in several blocks of 64KB when MBA is disabled (0-delay)

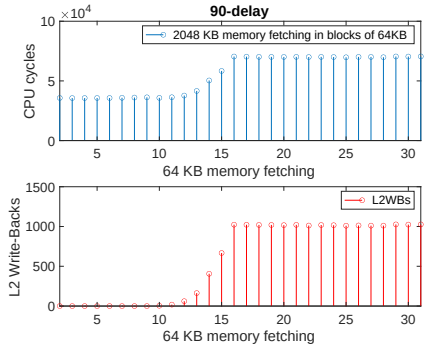


Figure 6.4. 2048 KB memory fetching in several blocks of 64KB when MBA 90-delay is enabled

Comparison with existing evaluations.

Finding 2: Instead of adopting generic workloads (e.g., RII), specific workloads (e.g., RI) have to be designed and executed to evaluate the performance of blackbox hardware controllers.
Finding 3: Existing works [124, 143] that adopted only RII as workload to estimate the regulation have the risk of an attacker capable, through the RI workload, to exceed the 50 % of the estimated regulation.

Memory time-sensitive Execution Environment. The regulations enforced by 70, 80 and 90 delays can be used to limit memory interfering cores located on the same hardware platforms, protecting a critical application in a time-sensitive execution environment. The maximum interference that co-located cores can run under these delay configurations corresponds to the RI workload. Even if the interfering cores try to run asynchronous memory requests to increase the accesses, the MBA delays LLC WBs as a consequence of delaying L2 WBs, which results in having the same total memory accesses (read and write) for the RI and WII workload under 70, 80, and 90 delays.

$$BW_{R^I;70:90} \approx BW_{W^{II};70:90} \tag{6.3}$$

Note that cores cannot cause LLC WB without causing L2 WB. However, even though simple and deterministic, this enforcement results pessimistic when L2 WBs do not cause LLC WBs, and therefore write, to the memory (Formula 4.2)

A safety memory time-sensitive execution environment

Finding 4: The regulations enforced by 70, 80 and 90 delays can be used to build a memory time-sensitive execution environment. The 80 and 90 delays limit the memory bandwidth of interfering cores to 75% and 50% of the synchronous memory bandwidth. In contrast, the 70 delay only avoids the overload of asynchronous requests due to L2 WBs.

These are important aspects to consider when dimensioning a system with MBA to assure a given memory bandwidth to a critical core.

Thus, in the multicore analysis performed in the next section, we evaluate the assured bandwidth when MBA limits the interfering workloads R^I only with 80 and 90 delay values, covering the significant cases of synchronous bandwidth allocation.

6.1.3 MBA Guarantees

We consider the following mixed criticality scenario. One critical task shares the hardware platform with three interfering cores. The critical task has the following execution model: periodically, the critical task (i) loads its working set from the memory to its private L2 cache (e.g., the working set might include the program, its state and inputs), (ii) it makes some computations, and, finally,(iii) it flushes its private caches updating the memory. To reproduce this scenario, the critical task has a WSS of 512KB, while the interfering cores reproduce the R^I workload.[3].

Note that the critical task is designed so that its working set does not rely on the LLC (Working Set Size (WSS) $<$ 1MB), and, due to the

[3]All four cores start with empty private caches, the critical task fetches 512KB reproducing the fetching phase, while the interfering cores reproduce RI by reading 768KB of memory. We choose a longer memory to be sure to generate seamless interference during critical task execution

exclusivity of the LLC, interfering cores should not cause the eviction of cache lines allocated in the private cache of the critical core.

In order to explore safety bounds for the initial memory fetching phase of the critical core, we measure the memory fetching latency $L_{k';(d_1,d_2,d_3)}$ of the critical core when the three interfering cores are delayed through $\{d_1, d_2, d_3\} \in D'$ MBA values. D' is the set of significant delay values identified in the previous section, i.e., 70, 80 and 90 delays that reduce the reachable bandwidth to 100, 75 and 50 per cent of the synchronous bandwidth, respectively. Figure 6.7 shows the observed $L_{k';(d_1,d_2,d_3)}$ (95-percentile), which corresponds to 10 different experiment setups.

To better interpret and predict the results, we adopted the model presented by Nowotsch & Paulitch [101]. The model assumes that the arbitration delay due to interference grows or is similar when the contention increases.

$$\frac{\delta_i}{i} \leq \frac{\delta_{i+1}}{i+1} \forall \in N^+, 1 \leq i \leq \left|\Pi_{||}\right|$$

In our quad-core configuration setup, $\Pi_{||}$ is four, representing the maximum number of contenders. To calculate the delay δ_i due to the contention of i contenders, we measure the latency of the critical workload when $i-1$ cores cause read interference and, then, we divide the latency among the requests. The arbitration delay is obtained by dividing the delay by the number of contenders $\frac{\delta_i}{i}$.

The red bars in Figure 6.7 show the predicted latencies obtained by applying the model of Nowotsch & Paulitch, given the capacities associated with the interfering cores. The observed latencies are lower than the predicted values and seem to work as safety bounds. To better explain the behaviour of the formula, we represent graphically the worst case they assume during the contention of the memory. Let's take as an example the first two configurations starting from the left of the Figure 6.7. Both configurations assume the interfering cores are regulated with 90-delay (first configuration) or 90-delay and 80-delay (second configuration). We expect the first configuration to cause a lower interference, as one of the three interfering cores has lower capabilities being throttled with the 90-delay rather than 80-delay. The model assumes the worst distribution of the requests is the one shown in Figure 6.5 and Figure 6.6 for the first and second cases, respectively. In Figure 6.5, the critical task π_1, during

the fetching of its Working Set, contends the memory for half of the requests with the three interfering cores π_2, π_3, π_4, which are throttled with a 90-delay, so they mostly can run 4191 requests instead of 8192, due to the throttling analyzed in the previous section. In Figure 6.6, one interfering core π_2 has a regulation of 80-delay instead of 90-delay, so it can run 6080 requests instead of 8192. Thus, in the worst case, the critical core contends the memory with all the interfering cores for the first half of requests, and then another quarter of requests instead interfere only with the second core ($\approx 8192 - 6080$).

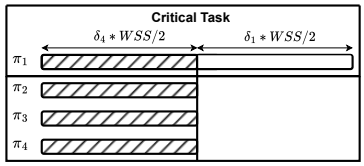


Figure 6.5. π_2, π_3, and π_4 are under 90-delay regulation

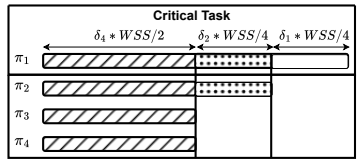


Figure 6.6. π_2 is limited by 80-delay regulation, while π_2, and π_3 are under 90-delay regulation

6.1.4 MQO as Interference Detector

The queue occupancy is the number of queued requests at a given instant of time. By monitoring memory queue occupancy, our goal is to estimate the detection time necessary to detect that more than k cores are co-accessing the memory.

To this aim, our experiments first measure the portion of time $P_{i,j}$ within a regulation period of one millisecond[4] in which the RPQ occupancy is greater than or equal to a given value i of requests when j cores contend the memory access. $P_{i,j}$ has a key role in the evaluation to understand if when we change the number of interfering cores j, the RPQ occupancy i assumes a different distribution.

To measure $P_{i,j}$, we make use of the "RPQ occupancy event" supported by the new series of Intel Scalable Processors as uncore performance event. This event registers the DRAM clock cycles the Read Pending Queue holds

[4]We set the regulation period to one millisecond similar to Memguard [152].

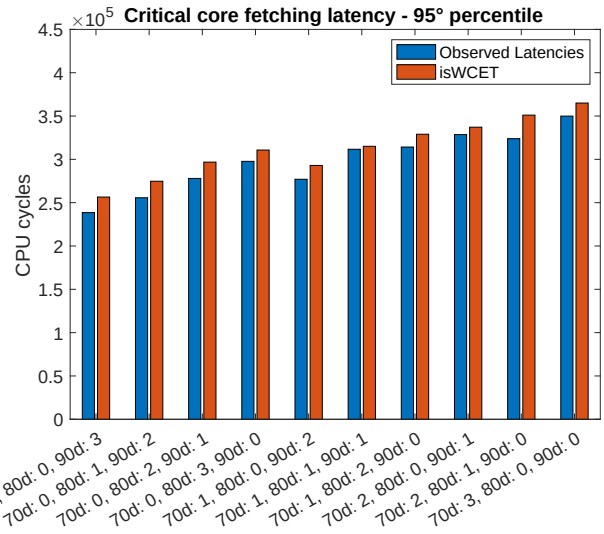


Figure 6.7. Observed critical memory fetching latencies (95p) versus isWCET predictive model (95p)

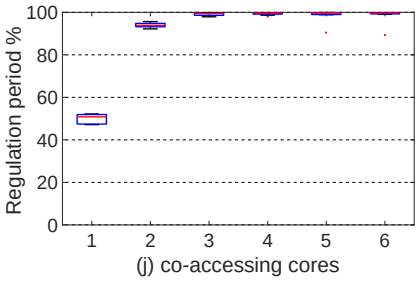
at least i requests, where i is a configurable threshold value. So, by reproducing memory interference from j co-accessing cores to the memory, we can monitor the number of cycles $Cycles_{i,j}$ in which the RPQ occupancy is at least i. Then, $P_{i,j}$ is obtained by normalizing $P_{i,j} = \frac{Cycles_{i,j}}{maxCycles}$[5].

Figure 6.8 shows $P_{i,j}$ when we change the number of co-accessing physical cores j from one to six ($j \in C, C = \{1; 2; 3; 4; 5; 6\}$). Every core reproduces only memory read operations (RII workload[6]). We set the RPQ occupancy threshold i in a multiple of eight ($i \in T, T = \{1; 8; 16; 24; 40; 47\}$) being enough to describe and include the total response variation. For instance, Figure 6.8 (a) plots the percentage of time in which the queue holds at least one request, representing the time in which the DRAM Channel is busy to service read requests.

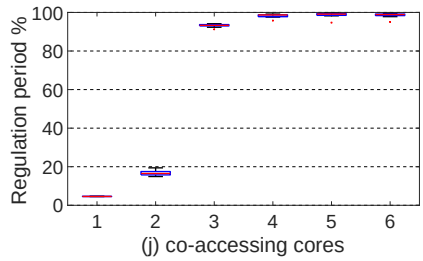
When two or more cores contend for memory access, the queue is busy servicing requests (Figure 6.8 (a)) for almost the whole regulation

[5]The monitoring counter can measure at maximum the regulation period (i.e., one millisecond) in terms of DRAM clock cycles.

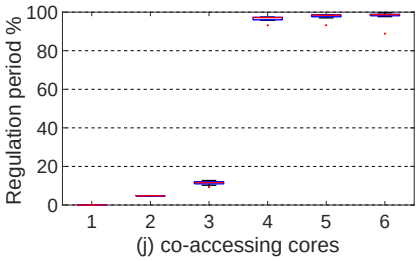
[6]Here we are not using the MBA so RI and RII produce the same number of memory reads within the regulation period



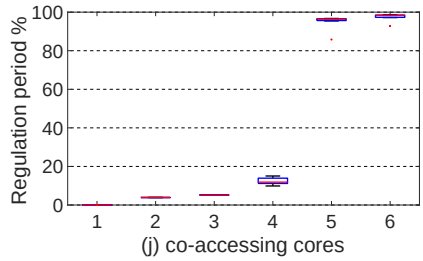
(a) (i=1) At least 1 request queued



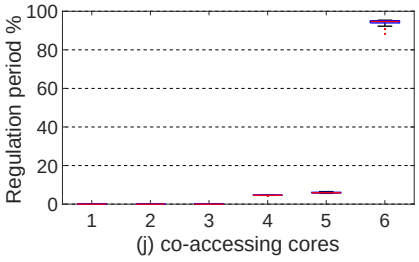
(b) (i=8) At least 8 requests queued



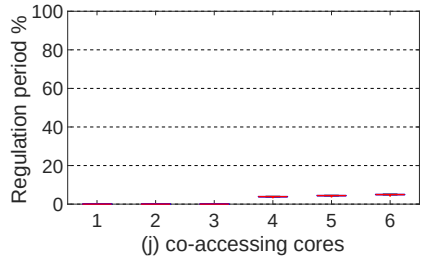
(c) (i=16) At least 16 requests queued



(d) (i=24) At least 24 requests queued



(e) (i=40) At least 40 requests queued



(f) (i=47) At least 47 requests queued

Figure 6.8. Portion of the regulation period in which the RPQ occupancy is at least i when j cores are co-accessing the memory

period. In such a condition, the memory controller services requests continuously and the memory throughput is saturated (around 280000 memory read operations in one millisecond). Note that, once exceeded by the two co-accessing cores, the throughput cannot represent the number of co-accessing cores, as it manifests a constant behavior,

However, while the throughput is constant once it is fully consumed, the queue occupancy grows. We expect that the queue occupancy does not grow indefinitely because the pending requests of the physical cores are limited. The other plots of Figure 6.8 confirm this assumption. From the results shown in Figure 6.8, we can observe that the queue occupancy reflects the number of the co-accessing core to the memory Visibly, the occupancy varies when more than 2, 3, 4, and 5 cores are co-accessing the memory in Figure 6.8 (b), (c), (d), (e). For instance, in Figure 6.8 (b)), setting j to two and three (i.e., two and three co-accessing cores), RPQ holds at least eight requests ($i = 8$) for one-fifth and for the entire regulation period, respectively.

Now we have obtained the distribution of RPQ occupancy in Figure 6.8 when applying different interfering cores, we can define the detection time, $DT_{i,k}$, i.e., the time necessary to detect that more than k cores are interfering:

$$DT_{i,k} = (1 - P_{i,k+1}) + P_{i,k}$$

where i is the RPQ occupancy threshold fixed on the queued requests.

For the sake of simplicity, we explain this formula through an example. We want to detect the interference degree corresponding to more than two co-accessing cores ($k = 2$). We set up the uncore PMC to monitor the cycles in which the queue at least holds eight requests ($i = 8$). Observing Figure 6.8 (b), when there are three physical cores, the queue holds at least eight requests for almost all of the regulation period. However, there is a slight difference ($maxCycles - Cycles_{8,3}$) because the ideal maximum throughput is still not fully achieved. Since we do not know the occupancy distribution in the regulation period, we need to assume the worst case.

In the worst case, after at most $(1 - P_{8,3})$ of the regulation period, the queue starts to hold at least eight requests when at least three co-accessing cores are active. Hence, under three co-accessing cores, the monitoring counter exceeds the cycles $Cycles_{8,2}$ reached from two-co accessing cores after almost $DT_{8,2} = (1 - P_{8,3}) + P_{8,2}$ of the regulation period. Hence, by

dedicating $DT_{8,2}$ of the regulation period in monitoring, we can detect if more than two cores are contending the memory. From a practical point of view (see Sec. 3.1), we can set an interrupt once the monitoring counter overflowed $Cycles_{8,2}$.

The second column of Table 6.3 shows the minimum detection time $min(DT_{occ})$ to detect more than k co-accessing physical cores to the DRAM channel by observing the variation of the occupancy. The third column provides the occupancy threshold tuned to reach the minimum detection time, while the fourth shows the detection time if we have adopted the variation of the throughput as the observable metric. If the detection time is one, the detection requires the entire regulation period to become unfeasible.

The results are promising: at most, one-fifth of the regulation period is enough for detection when the total memory bandwidth is consumed. In particular, only detecting more than two cores $k = 2$ requires one-fifth of the regulation period while detecting more than three, four, and five cores $k = 3, 4, 5$ requires one-sixth. Ideally and theoretically, we expect a detection time equal to zero when the memory throughput is fully consumed, e.g., we presume that some queue positions are never achievable when the number of the co-accessing core is limited. However, the ideal case does not include real sources of interference such as the DRAM Refresh Cycles in which the memory stops to service requests.

MQO as interference detector

Finding: By monitoring MQO, we can detect the number of co-accessing cores. Detecting more than two cores $k = 2$ requires a detection time corresponding to one-fifth of the regulation period while detecting more than three, four, and five cores $k = 3, 4, 5$ requires one-sixth.

Therefore, the occupancy reflects the number of co-accessing cores to the memory, even if the detection time can be further improved by locking other factors of interference such as the refreshes of the DRAM.

Table 6.3. Minimum Detection time to detect more than k co-accessing cores

k	$Threshold_{occ}$	$min(DT_{tp})$
1	1 reqs	0.5
2	8 reqs	0.94
3	16 reqs	1
4	24 reqs	1
5	40 reqs	1

6.2 Record and Replay in Xen

We perform a thorough experimental analysis of IRIS to respond to the following research questions.

- **RQ1: Accuracy.** What is the accuracy of IRIS at reproducing *VM behaviors* using recorded *VM seeds* via the proposed replaying mechanism?
- **RQ1: Efficiency.** What is the efficiency in submitting recorded *VM seeds* via our replaying approach in terms of the CPU time needed to execute related VM exits?
- **RQ3: Performance overhead.** What is the recording overhead in collecting the target info to build *VM seeds*?
- **RQ4: New Testcases - No manual Effort.** Is IRIS capable of generating new testcases without manual effort?

6.2.1 Methodology

We performed our experiments using a host machine with Intel Xeon i7-4790 @3.6Ghz, and 16GB RAM, running Linux kernel v5.10. We implemented IRIS on top of Xen hypervisor v4.16, running with *HVM* mode to enable hardware-assisted virtualization. Currently, the IRIS framework supports Intel VT-x hardware extensions. We run the IRIS *manager* in *Dom0*, and a guest workload is to be recorded and replayed in a *DomU*. This latter domain is our *test VM*, while a second *DomU* is mounted as the *dummy VM*. Each domain (both Dom0 and DomU) runs Linux kernel

v5.10 and is set up with a single vCPU pinned on a dedicated pCPU, 1 GB RAM, and 20 GB HDD. We impose a 1-to-1 vCPU/pCPU pinning for VMs to prevent as many as possible interferences (e.g., high rate of cache misses) between the different physical cores and obtain coverage data as clean as possible. To estimate the accuracy, we use the metrics provided by IRIS gathered during the execution of guest workloads.

Workloads. We use three workloads, i.e., OS-BOOT, CPU-bound, and IDLE, for our evaluation, as they fully cover the spectrum of VM exits registered across five distinct workloads: OS-BOOT, CPU-bound, MEM-bound, IO-bound, and IDLE. The *OS BOOT* workload makes the boot of an OS kernel; The *CPU-bound*, *MEM-bound*, and *I/O-bound* stress the CPU, memory, and IO subsystem, respectively, while the *IDLE* keeps idle the OS. Figure 6.10 summarizes the distribution of VM exits across the target guest workloads. For the sake of simplicity, we consider a sample trace of 5000 VM exits for each workload. We can notice that the *OS BOOT* workload results in VM exits that are mostly related to *I/O instruction* and *Control-register accesses* exit reasons. In the boot phase, the guest configures devices and the hypervisor is triggered to carry out operations for emulation, exclusive assignment, or I/O device sharing depending on the kind of virtualization approach used [1]. In the remaining workloads (i.e., *CPU-bound MEM-bound*, *I/O-bound*, and *IDLE*), almost 80% of VM exits are related to `RDTSC` instructions, which are related to kernel operations needed for timekeeping and implementing scheduling routines [130]. Further, the *IDLE* workload is characterized by some `HLT` VM exits, basically due to the implementation of the *idle loop* in the Linux kernel. Figure 6.10 also reveals that the hypervisor intervention is relegated to a few critical VM exits, which are common across different and heterogeneous workloads. For the sake of representativeness, we further analyzed benchmarks provided in [138], which exhibit similar VM exit distributions obtained above. For simplicity, the analysis provided in the next sections targets only the *CPU-bound* workload since it has several commonalities with the MEM-bound and I/O-bound experimented workloads.

OS BOOT, CPU-bound and IDLE workload. The *OS BOOT* workload, specifically refers to booting the Linux kernel, and it consists of about 520K VM exits until the OS presents the login screen to the user. Figure 6.9 details the VM exits distribution during *OS BOOT* workload

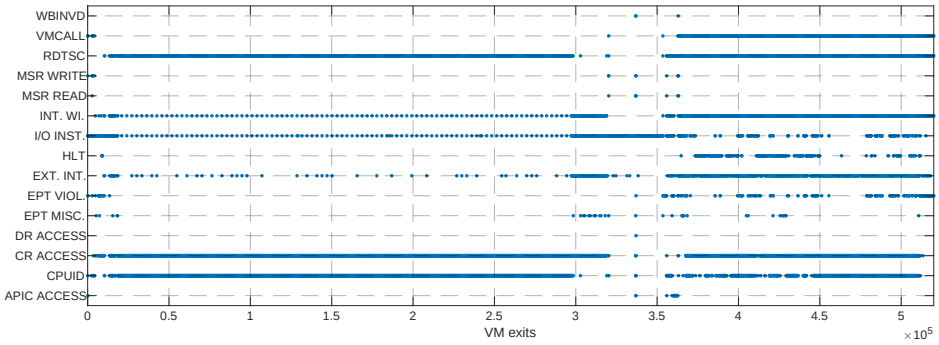


Figure 6.9. VM exit reasons distribution over time during *OS BOOT* workload.

over time. Note that IRIS is capable of recording all the VM exits occurring during the boot sequence. The distribution includes a sequence of VM exits (the first 10K) that are related to the BIOS emulated by Xen [141], which is not part of the OS BOOT we want to characterize. Given this, our OS BOOT trace of 5000 VM exits starts after the last BIOS VM exit. The *CPU-bound* workload includes 5000 VM exits triggered during the execution of CPU-intensive operations (e.g., compute Fibonacci sequence, matrix operations, etc.). *IDLE* workload includes 5000 VM exits triggered during the OS idle loop.

6.2.2 Accuracy

We analyze how accurate is IRIS in the automatic recording and replaying of *VM behaviors* using key metrics mentioned in section 4.2, i.e., code coverage and the pair {VMCS field, value} written. For this purpose, we aim to reveal the difference between metrics gathered both in the recording and replaying phases for each VM seed obtained during the execution of target workloads described in section 6.2.1. Note that we use the same VM snapshot as the starting state for the record and replay phases to unbias the accuracy evaluation. Regarding accuracy in terms of code coverage, Figure 6.14 shows the results across the target workloads. The recording curve identifies the cumulative coverage trend for *VM seeds* recorded, in which we evaluate the unique lines of code discovered dur-

ing VM exit handling, for each *VM seed*. Instead, the replaying curve identifies the cumulative coverage trend for the replaying of the same VM seeds. The code coverage fitting at the end of replaying *VM seeds* is 99.9%, 92.1%, *and* 98.9% for *OS BOOT*, *CPU-bound*, and *IDLE* workloads, respectively.

Despite we achieved high accuracy in code coverage, we further analyze the remaining differences qualitatively. Figure 6.15 shows the code coverage differences across the three target workloads, which we clustered by VM exit reasons. The code coverage data may be affected by sources of non-determinism due to asynchronous events that interrupt the hypervisor (in VMX root mode) during the VM exits handling. The minimum code coverage difference ranges from 1 to 30 LOC. By analyzing such cases, we point out that such differences are related to the *local Advanced Programmable Interrupt Controller (*`"vlapic.c"`*)*, *interrupt handling* (`"irq.c"`), and *virtual timer* (`vpt.c`) Xen components. We can treat such coverage differences as noise to filter out.

We also investigate the cases when the code coverage differences are greater than 30 LOC. The frequency of these cases (filtering the repeated VM seeds in a workload) is 0.36%, 0.18%, and 1.16% for *OS BOOT*, *CPU-bound*, *IDLE* workloads, respectively. These differences refer to the *HVM*

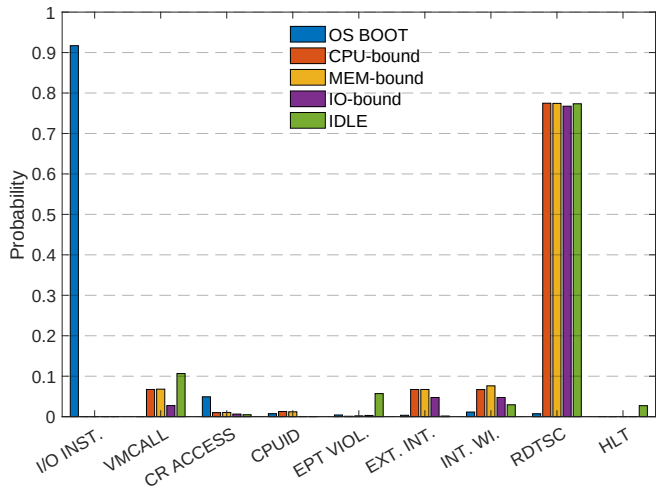


Figure 6.10. VM exit reasons distribution over different target workloads.

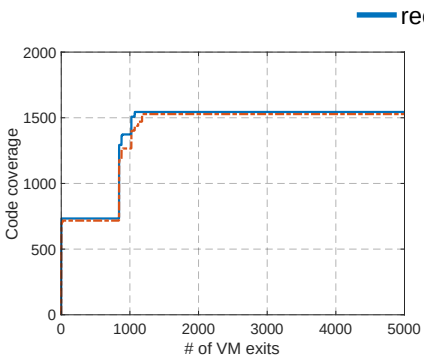


Figure 6.11. OS BOOT

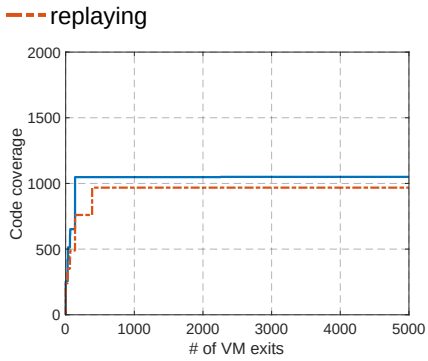


Figure 6.12. CPU-bound

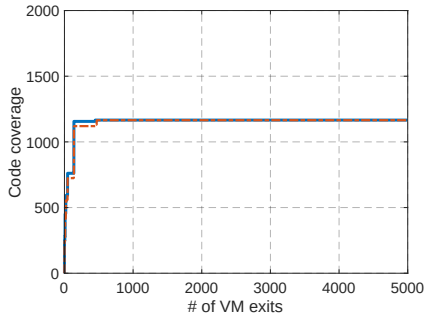


Figure 6.13. IDLE

Figure 6.14. Cumulative code coverage across *OS BOOT*, *CPU-bound*, and *IDLE* workloads.

instruction emulator (`"emulate.c"`) and *VM exit handler* (`"intr.c"`, and `"vmx.c"`) Xen components. This behavior can be due to the recorded VM seeds that are linked to memory-related VM exits. For example, VMCS fields like *Global* and *Local Descriptor Table Registers* (GDTR and LDTR) include references to the memory of "exited" guest VM. Such values can be dereferenced by the hypervisor during exit handling.

IRIS accuracy

Finding: *IRIS* is accurate to automatically generate (record) seeds to reproduce real VM behaviors, showing a fitting of code coverage ranging between 92.1% and 100% in our settings, compared to real guest execution.

We further evaluate the IRIS accuracy, by focusing on the `VMWRITE`s metric, which provides a more fine-grained measure of actual VM state changes (guest-state area) from the point of view of the VMCS and VMX operations.

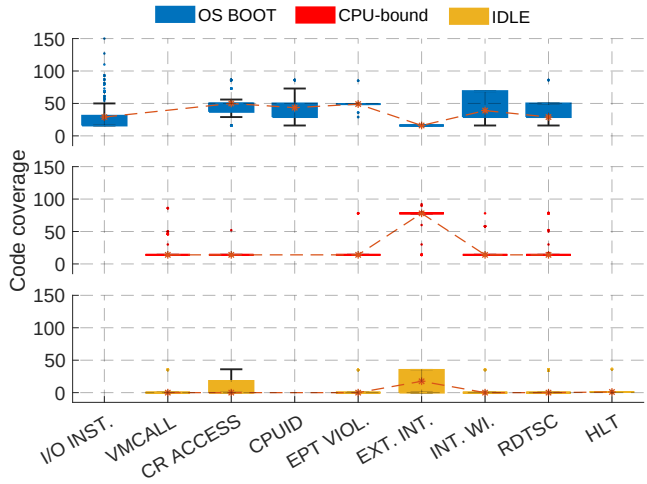


Figure 6.15. Code coverage differences by VM exit reason across targeted workloads.

Focusing on the entire *OS BOOT* workload (see Figure 6.9), the OS switches operating modes and CPU states several times. In that case, the fitting on the executed `VMWRITE`s on the VMCS guest-state area is 100%. Indeed, Figure 6.16 shows an example for `VMWRITE` operations both recorded and replayed against the control register zero (i.e., `CR0`). Each of the modes represents a set of states held by the CR0 register.

In particular, *Mode1* and *Mode2* indicate *real mode* and *protected mode*, respectively. *Mode3* specifies *protected mode* with *paging enabled*,

Mode4 includes *Mode3* with alignment checking performed, *Mode5* includes *Mode4* with test of task switch flag, *Mode6* includes *Mode4* and caching enabled, *Mode7* includes *Mode5* and caching disabled. Results suggest that the proposed recording approach allows us to generate seeds that closely follow real *VM behaviors* of guest execution. Finally, we run an experiment to provide evidence that replaying recorded VM seeds allows reaching the same hypervisor state as in the real guest execution. We replay *CPU-bound* and *IDLE* workloads from a *i)* VM state without booting the OS, and from a *ii)* VM state reached by replaying the recorded *OS BOOT* VM seeds. In the former case, the *dummy VM* crashes (Xen logs: `bad RIP for mode 0`, where `mode 0` is *Mode1* in Figure 6.16), while in the latter case both the *CPU-bound* and *IDLE* workloads complete.

6.2.3 Efficiency

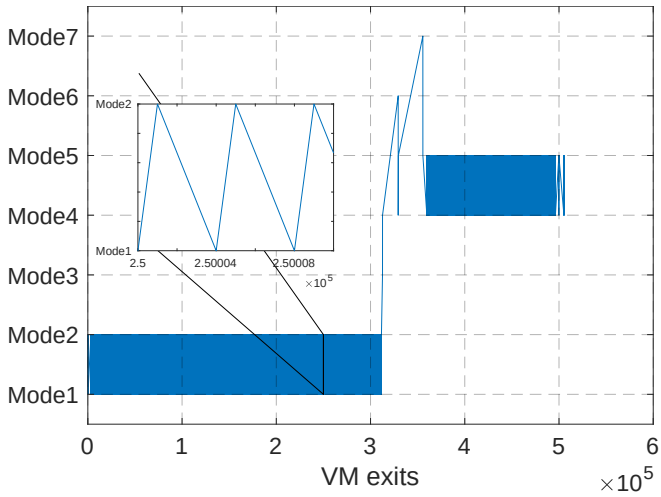


Figure 6.16. Operating modes and virtual CPU states across VM exits during *OS BOOT* workload.

Seed submission is fundamental in developing fuzzers since it heavily impacts fuzzing efficiency [161]. To this end, we estimate how *IRIS* is efficient in reaching VM states compared to the real guest VM execution. We performed this analysis across workloads described in subsection 6.2.1,

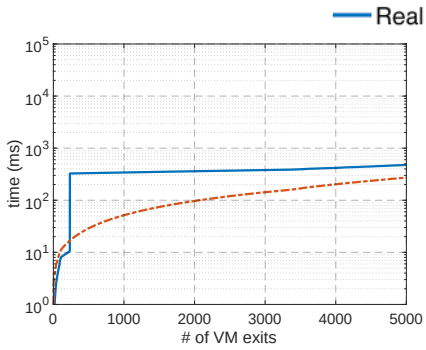


Figure 6.17. OS BOOT

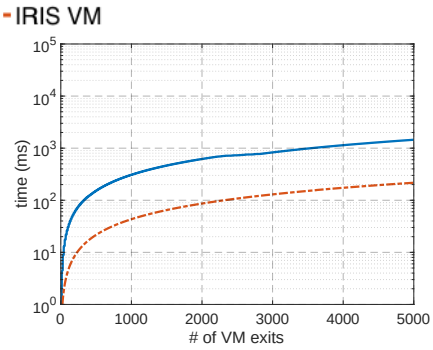


Figure 6.18. CPU-bound

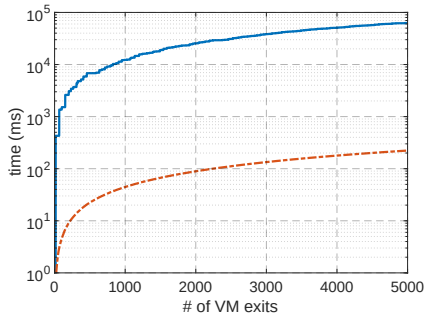


Figure 6.19. IDLE

Figure 6.20. Performance in submitting *VM seeds* across *OS BOOT*, *CPU-bound*, and *IDLE* workloads.

by running the experiments 15 times for statistical significance purposes, obtaining the same results with a high level of confidence (p-value < 0.05).

Figure 6.20 shows the time needed to submit VM seeds by real guest VM execution (see *Real VM* in Figure 6.20 and by using the IRIS replaying mechanism (see *IRIS VM* in Figure 6.20). The results show that our replaying mechanism can replay real guest VM behaviors efficiently, with a percentage decrease of 42.5% ($0.27s$ vs $0.47s$), 85.4% ($0.21s$ vs $1.44s$), and 99.6% ($0.22s$ vs $62.61s$), for *OS BOOT*, *CPU-bound*, and *IDLE* workloads respectively. It is worth noting that the *OS BOOT* exhibits the main differences in the first 1000 VM exits, in which the kernel OS spends time running guest operations that do not require the hypervisor intervention.

These non-sensitive instructions delay substantially the subsequent VM exits that eventually need to be handled by the hypervisor; thus, there is a non-negligible latency in discovering the same coverage compared to the replayed workload. In general, the throughput of our replaying mechanism is roughly linear, as also confirmed by the time distributions to replay *CPU-bound*, and *IDLE* workloads (see Figure 6.18 and Figure 6.19). These workloads require less hypervisor intervention (VM exits handling), thus the IRIS replaying is even better to discover the same coverage as real guest execution in less time, with a speedup factor of 6.8× and 294× for *CPU-bound* and *IDLE* workloads, respectively.

IRIS efficiency

Finding: *IRIS* can replay recorded seeds to reach a valid VM state, with a time improvement from 42.5% to 99.6% compared to real guest execution.

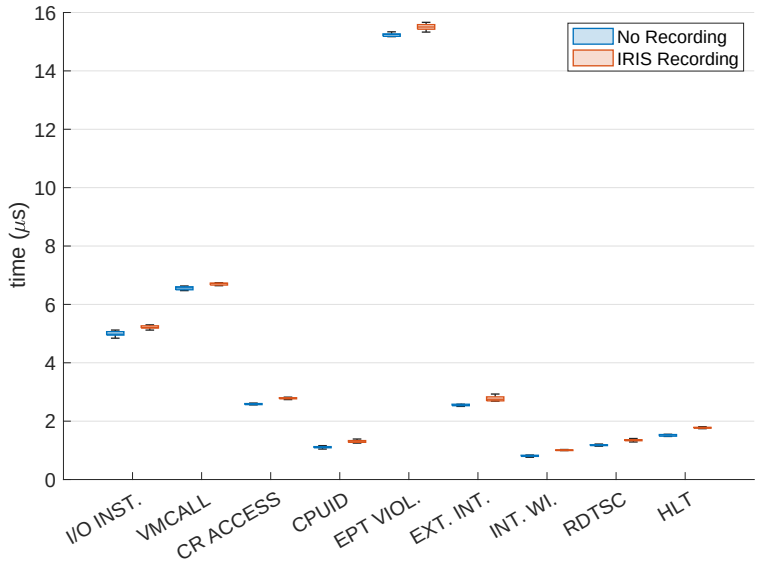


Figure 6.21. The temporal overhead, for each *VM exit*, induced by IRIS recording.

In addition, we also measure an *ideal replaying throughput* to have an upper bound for estimating the maximum replaying efficiency. We computed this value by running the preemption timer VM exits in the same number of the VM exits needed per workload (i.e., 5000 VM exits), and measuring the time needed to handle them. We obtained $0.1s$ ($\sim 350M$ CPU cycles for our testbed), which means $50K$ VM exits/s. Comparing to this *ideal replaying throughput*, we obtained a percentage difference of 63% (18.518 VM exit/s), 52% (23.809 VM exits/s), and 55% (22.727 VM exits/s) for *OS BOOT*, *CPU-bound*, and *IDLE* workloads, respectively. However, such difference does not contemplate the logic behind replaying mechanisms, but it is useful to make new room for improvements.

6.2.4 Performance Overhead

About the overhead induced by the IRIS recording process, we first analyze the target workloads (i.e., *OS BOOT*, *CPU-bound*, and *IDLE*) to reveal the temporal overhead for each VM exit. We run the workloads 10 times, taking the median values of time needed by the Xen *VM exit handler* to serve a specific VM exit. Figure 6.21 shows the boxplots across the VM exits handled during workload execution, with and without IRIS recording activated. The results show a very small overhead, ranging from $1,02\%$ to $1,25\%$ percentage increases in the best and worst cases, respectively. Concerning the IRIS memory overhead induced during recording/replaying, we need to consider the size of the *VM seed* for each VM exit and the reads/writes performed on the VMCS. In the worst case, we experimented 32 VMREAD/VMWRITE operations on the VMCS across all the target workloads, obtaining a VM seed size of 470 bytes for each VM exit. The current implementation of the IRIS recording pre-allocates a heap memory equal to the VM seed size in the worst case (i.e., 470 bytes) for each VM exit to be recorded. Instead, the IRIS replaying allocates exactly the needed heap memory for each VM seed recorded since we know in advance the number of VMREAD/VMWRITE operations performed on the VMCS.

Performance overhead

Finding: IRIS shows a very small overhead during the recording phase, ranging from $1,02\%$ to $1,25\%$ percentage increases in the best and worst cases, respectively.

6.2.5 New Testcases - No Manual Effort

We build a proof-of-concept (PoC) to show the potential of using IRIS to effortlessly run fuzzing experiments on Xen, as an example of a hardware-assisted solution. The aim is to show that the PoC fuzzer can discover new code coverage and detect anomalous hypervisor behaviors. The fuzzing logic includes *i)* adopting the IRIS replay mechanism to move into valid VM states by utilizing *VM seeds* obtained during the recording of target workloads (i.e., *OS BOOT*, *CPU-bound* and *IDLE*), and ii) mutating a specific VM seed by corrupting VMCS fields and GPR. According to the hypervisor fuzzing literature, we consider the guest VM untrusted. Specifically, the guest VM operations affect directly the VMCS (guest state) and indirectly the hypervisor control flow.

Test cases. The test cases we plan are characterized by the following factors: *i)* the replayed *VM behavior* W of target workloads, *ii)* a target *VM seed* $VMseed_R$ took randomly within the *VM behavior*, and *iii)* the *VM seed* area $A = \{VMCS, GPR\}$, of $VMseed_R$, to mutate.

Each test case starts from an initial VM state s_0 of W (i.e., by starting the VM). Next, the fuzzing logic uses IRIS to replay the *VM behavior* until $VMseed_R$ is reached, to move to the linked VM state (see s_1 in Figure 6.22). At this point, the fuzzer mutates the $VMseed_R$ by generating M (set equal to 10000) mutated versions, defining the *fuzzing sequence* as $C(VMseed_R)_1, ..., C(VMseed_R)_M$, which moves the hypervisor into an unseen state s_M. Such a sequence can be submitted via the IRIS replay mechanism, according to chosen *mutation rules*, as depicted in Figure 6.22.

Mutation rules. The structure of test cases described above includes the *fuzzing sequence* to be submitted. These mutations focus on a specific *VM seed* area (i.e., VMCS or GPR) of the $VMseed_R$. The mutation rule we adopt includes a *single bit-flip* in *VM seed* area. Specifically, the fuzzer randomly selects a VMCS field or a general-purpose register and then bit-flip the value (e.g., `0xFFFFFFF0` to `0xFFFFFFF1`).

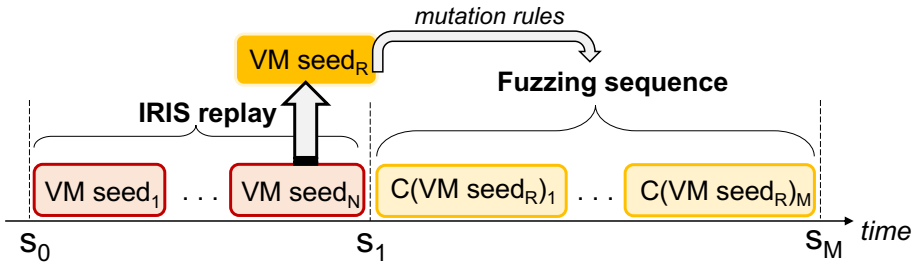


Figure 6.22. Test cases structure in the IRIS-based fuzzer prototype.

Failure modes. By using scripts that analyze hypervisor behavior and logs, the PoC fuzzer can detect failures occurring during the execution of test cases, that we classify as hypervisor or VM crashes. These can be due to double faults, an invalid operation, page faults, etc. In these cases, the test case, as well as the submitted VM seeds, are saved for further investigation with the aim of crash analysis to reveal potential bugs in the source code.

Results. Table 6.4 shows the new code coverage discovered by running planned test cases, as explained earlier. The code coverage we consider as the baseline is discovered by the single $VMseed_R$, while each cell (i.e., a test case) of Table 6.4 shows the percentage increase of code coverage discovered by submitting the *fuzzing sequence*. In all tests, we can observe newly discovered coverage, with a significant increase in the *OS BOOT* case, due to the complexity of the workload itself. Note that code coverage information can be retrieved for each VM seed submitted. Regarding failures, we observed VM or hypervisor crashes in respectively 1% and 15% of the tests when the VMCS is mutated. A small number of VM crashes has also been observed when mutating the GPR together with a `CR ACCESS` (as exit reason). In all other cases, the hypervisor is not affected by the mutation. The results show how the IRIS-based fuzzer PoC can discover new hypervisor code coverage and crashes, by planning a few test cases with a naive mutation rule. The manual effort in building fuzzing seeds is negligible since we obtained them by leveraging the IRIS recording mechanism. Furthermore, seed submission is done by reusing the IRIS replay mechanism with no other external tools.

Table 6.4. New code coverage discovered across test cases by using IRIS-based fuzzer prototype.

Exit Reason	**OS BOOT**		**CPU-bound**		**IDLE**	
	VMCS	GPR	VMCS	GPR	VMCS	GPR
EXT. INT.	+122%	+76%	+7%	+3%	+7%	+3%
INT.WI.	+115%	+61%	+6%	+3%	+6%	+3%
CPUID.	+124%	+71%	+14%	+2%	-	-
HLT	-	-	-	-	+7%	+2%
RDTSC	+120%	+69%	+17%	+2%	+17%	+2%
VMCALL	-	-	+18%	+3%	+16%	+3%
CR ACC.	+10%	+63%	+13%	+2%	+10%	+2%
I/O INST.	+8%	+58%	-	-	-	-
EPT VIOL.	+22%	+50%	+13%	+1%	+18%	+10%

Chapter 7

Work-in-progress: transparent replay

In this Chapter, we present the first evaluations performed on HyRo implementation, especially those regarding replay accuracy.

In order to evaluate the effectiveness and efficiency of the work-in-progress HyRo implementation, we respond to the following research question.

Accuracy. Is HyRo replay accurate to reverse hypervisor states of different hypervisors?

7.0.1 Methodology

Environment setup. We conducted the experiments on a host machine featuring an Intel Xeon i7-4790 processor @3.6 GHz, and 16GB RAM, operating on Linux kernel v5.19.17. Each virtual machine (VM) is configured with 2 vCPUs and 2GB of memory, with each L2 guest VM allotted 1 vCPU and 1GB of memory. Importantly, each vCPU maintains a one-to-one pinning with a dedicated physical CPU core.

Metrics. Contrary to IRIS, we do not adopt code coverage to evaluate the accuracy for two main reasons. Without hardware support, tracing the code coverage requires the instrumentation of the target hypervisor to show acceptable performance, resulting not a portable solution[1]. Second,

[1]The alternative would be trapping every single instruction of the target hypervisor

code coverage is more subject to be affected by noise (not controllable by the guest) affecting accuracy results. As a consequence, instead of code coverage, we compare the input/output operations to the VM state performed by the hypervisor during its intervention in the nominal and replied execution. In particular, we compare the read and write operations performed on the VMCS and the output general purpose registers registered after the hypervisor intervention. We call this metric VM_{state} *coverage*. Hence, we define the accuracy of our replay mechanism as the ability to faithfully reproduce a previously recorded hypervisor behavior, with VM_{state} *coverage* serving as the metric to quantify this accuracy.

Mismatch cases in VM_{state} coverage

Note: Regarding VMread and VMwrite operations to the VMCS, we may observe different cases comparing the recorded and replied execution. We list these cases in the table illustrated below. We may observe *case 1* if a specific field of the VMCS is not read or written in both executions. We have *case 3* and *case 4* if the operation has been observed only in the recorded or replied execution, respectively. We have *case 2* when the identical operation has been observed with the same parameters. Instead, *case 5* describes the case in which the same field of the VMCS has been accessed, but with a different read or written value. We summarize these cases with two metrics, the *Fitting* and *Divergency* metrics, illustrated in the last two rows of the table.

	Case 1	**Case 2**	**Case 3**	**Case 4**	**Case 5**
Record	NaN	X	X	NaN	X
Replay	NaN	X	NaN	X	Y
Occurrences	N1	N2	N3	N4	N5
NREC	N2+N3+N5				
NREP	N2+N4+N5				
Fitting	N2/NREC				
Divergency	(1-N2)/(NREP))				

We specialized VM_{state} *coverage* into two dimensions to have better visualization during the representation of the accuracy: (1) `Fitting`: the percentage of operations of the replied execution that were observed during the recorded execution, e.g., having registered 10 VMREADS during the recorded execution, and having observed 5 of these VMREADS dur-

ing the replied execution, the fitting result is 50 %. (2) `Divergency`: the percentage of operations of the replied execution that were not observed during the recorded execution, e.g., having registered 15 VMREADS during the replied execution, and only five of these have been observed during the recorded execution, the divergency result is 67%, i.e., $10/15 = 0,67$. So, the *fitting* and *divergency* of VM_{state} *coverage* have optimal values if they are equal to 100% and 0%, respectively.

We obtain the efficiency of HyRo as the time in terms of CPU cycles spent for replaying hypervisor behaviors.

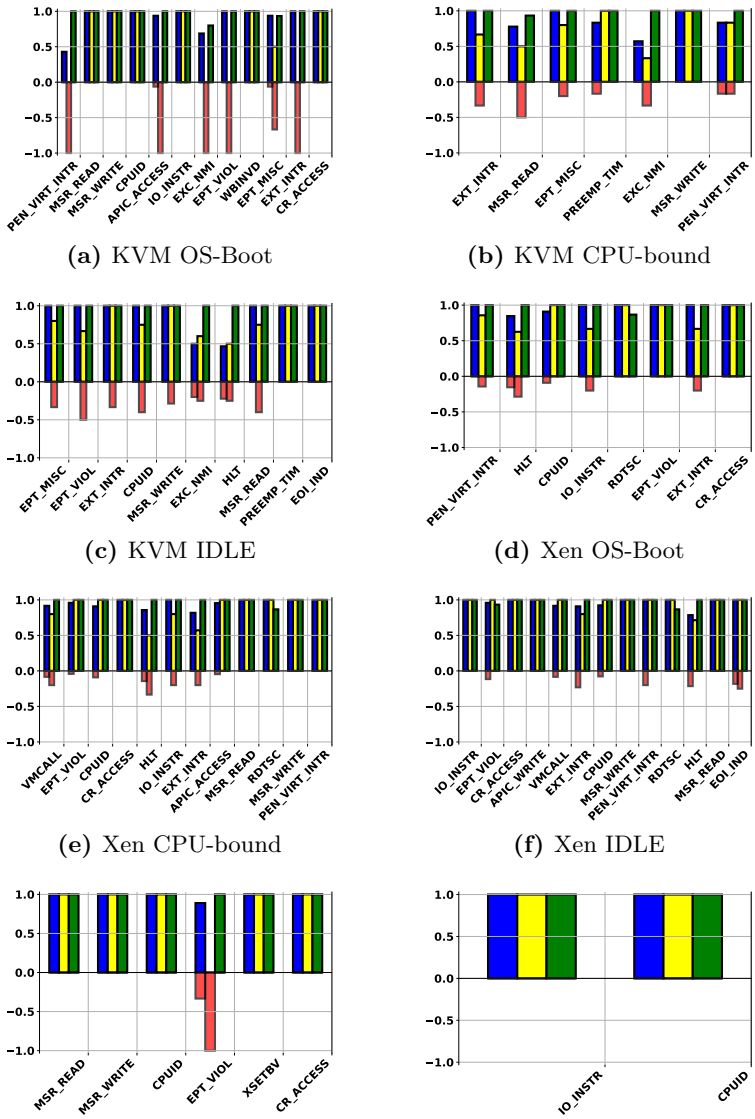
Workloads We use three workloads (the same workloads adopted for IRIS evaluation): (i) the booting process of a Linux OS kernel, denoted as `OS Boot`; (ii) maintaining the Linux OS in an idle state, referred to as `Idle`; (iii) CPU-intensive operations, denoted as `CPU-Bound`. The `OS Boot` workload is distinguished by hundreds of thousands of VM exits and extends until the initiation of the user login screen. The initial phase of the `OS Boot` workload includes operations that are not under the control of the guest. These operations are controlled by the hypervisor to configure the VM. Afterwards, the guest OS bootloader is loaded, which prepares the necessary components for the OS startup, i.e., it configures devices, constructs its page table, and so forth. The `CPU-bound` workload includes CPU-intensive tasks, such as computing the Fibonacci sequence, matrix operations, etc. Instead, the `Idle` workload is the guest operations executed during the OS login screen.

7.0.2 Accuracy

Figure 7.1 represents the average fitting and divergency of VM_{state} coverage observed across the target hypervisors and the selected workload, especially the VM_{state} coverage regarding the fitting of accesses to the VMCS (VMreads and VMwrites operations) and to the GPR (GPR values after hypervisor intervention). We avoided to show the accuracy for the CPU workload in Jailhouse, as we observed its execution does not cause any VM exits during the recording execution.

Analysis of the observed results. To better represent the results, we grouped the exit reasons based on their nature:

- **CPU Instructions:** These include CPU instructions (that do not



(a) KVM OS-Boot **(b)** KVM CPU-bound

(c) KVM IDLE **(d)** Xen OS-Boot

(e) Xen CPU-bound **(f)** Xen IDLE

(g) Jailhouse BOOT **(h)** Jailhouse IDLE

Figure 7.1. Fitting of VMreads (●), VMwrites (●), GPR (●), with their respective divergencies, represented as negative values (●)

fall in the other categories) that the hypervisor may choose to trap (e.g., MSR READ, MSR WRITE, CPUID, WB INVD, CR ACCESS, RDTSC, HLT), represented in Table 7.1 and Table 7.2 as "CPU instruction".

- **Memory-mapped accesses:** Memory-mapped accesses are those guest accesses to memory areas that cause the hypervisor intervention, represented in Table 7.1 and Table 7.2 as "Memory related". The observed exit reasons that fall in this category are EPT VIOL, IO INSTR, and EPT MISC.

- **Apic emulation:** The virtualization of the APIC on the current Intel architecture is not accelerated by hardware features, so the hypervisor traps and emulates every access to the APIC (i.e., APIC ACCESS, APIC WIRTE), to virtualize its behavior. Exit reasons due to the APIC are represented in Table 7.1 and Table 7.2 as "APIC".

- **Asynch Events:** We group in this category all the exit reasons that are linked with asynchronous events, represented in Table 7.1 and Table 7.2 as "Async". For instance, in our experiments, we observed PREEM TIM, PEN VIRT INTR, EXT INT, EOI IND

- **VMX Instructions:** We group in this category all the exit reasons that are related to new instructions introduced with Intel-VTx. However, in our experiments, we observed only VMCall as exit reasons of this category, so we do not represent this category in Table 7.1 and Table 7.2.

By the results of Table 7.1 and Table 7.2, we observe that: (1) the fitting and divergency of VMreads (i.e., the input to the hypervisor intervention) performs better than the VMwrites (i.e., the output from the hypervisor intervention), especially for the exit reasons falling in the API, Async, and Memory related categories, (2) Hyro replay performs better for the "CPU Instruction" category than the other categories.

Sources of Inaccuracy. We conduct a qualitative analysis of the variability to gain more insights about the origins of errors, in order to address these in future works. Below, we summarize the main causes of inaccuracy identified in the current implementation of Hyro from our qualitative analysis:

Table 7.1. Results for VMREAD operations

	Jailhouse		XEN		KVM	
	Fitting	Divergency	Fitting	Divergency	Fitting	Divergency
CPU Instruction	1	0	0.9523	0.0549	0.9791	0.0183
Memory related	1	0	0.9833	0.0258	0.9883	0.01
Async	-	-	0.95	0.0758	0.783	0.051
APIC	-	-	0.975	0.02	0.93	0.06

Table 7.2. Results for VMWRITE operations

	Jailhouse		XEN		KVM	
	Fitting	Divergency	Fitting	Divergency	Fitting	Divergency
CPU Instruction	1	0	0.9314	0.0506	0.9248	0.1525
Memory related	1	0	0.91	0.0633	0.8075	0.4467
Async	-	-	0.8133	0.0867	0.6625	0.44
APIC	-	-	1	0	0.465	1

1) Memory Dependency. The current implementation cannot record and replay hypervisor accesses to VM memory during hypervisor interventions. For example, in cases of EPT (Extended Page Table) violations, the hypervisor performs accesses to the guest memory to retrieve the instruction that caused the VMexit, allowing it to decode information not available in the VMCS, essential for completing emulation. This scenario is evident in Jailhouse during boot workload where we do not use guest snapshots. However, replay accuracy can improve when starting from a VM snapshot with the correct memory layout. For instance, if the snapshot captures the guest's program already loaded, the hypervisor locates the instruction correctly at the program counter, explaining why the EPT VIOL replay accuracy in Xen OS boot performs well.

2) Statefulness. Even if hypervisor accesses to VM memory are limited, inaccuracies in one VM exit can propagate if there are state dependencies across VM exits. For example, if a state change occurs during an EPT violation and the replay fails to update the hypervisor state accurately, subsequent VM exits dependent on that state may inherit inaccuracies. This explains why different VM exits show varying accuracy across workloads.

3) Asynchronous Events. The hypervisor may use asynchronous

events to control and interact with guest execution. For example, it might inject an NMI exception (EXC_NMI), set a preemption timer (PREEMP_TIM) to regain CPU control or respond to an emulated device signalling an event with an external interrupt (EXT_INT). These interventions vary for two primary reasons: (1) For NMI and PREEMP_TIM, hypervisor intervention may depend on state variables accessed by the host, not the guest CPUs, which are outside our recording scope. (2) Events such as EXT_INT are indirect consequences of other VM exits, such as VM exits (e.g., EPT_VIOL) related to MMIO guest accesses. Here, timing is crucial; an emulated device might require processing time before interrupting the guest, affecting the replay injection timing.

Chapter 8

Related Work

We report the related works about memory access isolation and hypervisor testing. Section 8.1 classifies the approaches adopted in the real-time community to apply memory bandwidth regulation. This section highlights where the Intel MBA is located and what are its main advantages: better resource utilization and fine-grained throttling. Instead, Section 8.1 discusses the works related to hypervisor testing, evidencing the need for a framework able to generate testcases without manual effort for testing the security of CPU virtualization.

8.1 Memory Access Isolation

In multicore processors, single task worst-case CPU utilization/execution time (Worst Case Execution Time (WCET)) estimation is highly pessimistic, especially, without partitioning the memory bandwidth [109]. Some approaches develop completely new specialized real-time processor architecture designs to tackle such issues. For example, Edwards and Lee [36] suggested the Precision Timed (PRET) architecture to deliver timing predictability and repeatability. Reineke et al. [112] proposed a new DRAM controller design for the PRET architecture to improve timing predictability and isolation between tasks at DRAM level. Contrarily, this thesis focuses on Commercial Off-The-Shelf (COTS) multicore processors, and thus, making changes to existing hardware architecture is not applicable here.

Existing work proposes some methodologies to analyze the impact of shared resources (including memory bandwidth) on the WCET of tasks in COTS multicore processors. For example, Pellizzoni et al. [109] show how to upperbound the task delay due to memory contention. Chattopadhyay et al. [27] present a unified WCET that considers shared caches, on-chip interconnect, and other micro-architectural components such as instruction pipeline and branch predictor. Such methods involve searching vast state spaces to include all possible task interactions. As a result, they suffer from enormous computational complexity.

Interference regulation approach types in existing literature can be broadly divided into the following types: Execution Model, Resource Limitation, and Resource Reservation. Some works also explore the combination of some of these approaches. Intel MBA [64] is a hardware-based solution of the Resource Limitation approach. Below, we review these main approaches.

8.1.1 Execution Model

In [114], authors propose decomposing tasks into a fixed sequence of superblocks consisting of three phases: acquisition, execution, and replication. Memory accesses are only possible in the acquisition and replication phases, and the acquisition or replication phases of simultaneously executing tasks cannot overlap to avoid resource contention. This model requires tight static analysis, and minor design changes can significantly increase a task's worst-case response time.

Houdek et al. [60] provided building blocks for implementing the Predictable Execution Model (PREM) [108] on an ARM-based multicore processor. PREM splits task jobs into two non-preemptible intervals: 1. Predictable interval with no system calls and interrupts; 2. Compatible interval with no special provisions. The predictable interval has two phases: a memory phase to prefetch all task data and instructions into private caches and an execution phase to perform the required task computation predictably without resource contention. Tasks need code instrumentation to mark task sections for execution in predictable intervals and a specially designed compiler to use the PREM model. Yao et al. [147] introduced a memory-centric scheduler to improve performance in a TDMA-based memory arbitration and considered a PREM model. When the TDMA

scheduler grants a memory slot to a core, the memory-centric scheduler reduces the priority for all active jobs in the execution phase on this core as compared to those in the memory phase on the same core.

The sliced execution model [22] divides the CPU execution time into multiple slices of two types: 1. Execution slice: a core executes a task based on prefetched instruction and data; 2. Communication slice: the core flushes all data from the previous execution slice to the DRAM and prefetches the instruction and data required for the next execution slice. A special toolset is required to generate the sliced architecture.

Biondi et al. [21] use the Logical Execution Time model [76] that decouples a task's CPU execution phase and the communication phase. The authors restrict the memory accesses of each task (prefetching and writing to memory) to precise time windows (communication phase) located at the beginning of a task's period, thus avoiding memory contention by design. As noted by the authors, such a model can have priority inversion as the LET communication phase for a low-priority task can delay the execution of a high-priority task.

Such models often poorly utilize the resources [71]. Execution models in general are too invasive for Mixed-Criticality Systems. They require code or compiler modifications that make it challenging to transition single-core legacy applications to COTS multicore processors and can increase the re-certification costs.

8.1.2 Resource Limitation

Approaches based on resource limitation free the non-critical tasks to be structured as execution models. They limit the number of accesses within a regulation period for non-critical tasks without modification to the original program. The limitation enables Incremental Development and certification [44] to define and reproduce the limited maximum interference for the memory fetching phase (the Acquisition phase) of the critical tasks. These solutions are usually based on a throttling policy to stop the processing element access in case of excessive requests and a monitoring policy for counting the issued requests. Notably, Bellosa [17] is among the first to perceive the potential of hardware performance counters to monitor shared resource accesses. The idea is to capture only the interesting available events on each processing element (PE), such as

last-level cache hits and references, to extract the number of LLC misses (representing synchronous accesses). Subsequently, LLC misses are supported as non-architectural events on modern COTS processors and are also used to monitor PE accesses [151, 8, 45, 48]. As a throttling policy, Bellosa [16] inserts idle loops inside the TLB-miss handler. Yun et al. [151] analytically calculate memory throttling parameters to guarantee memory bandwidth to a safety-critical task executing on one core while limiting the impact on non-critical tasks executing on all the other cores. They count LLC miss performance counter events to determine the number of memory accesses by a core. Nowotsch et al. [101] proposed the concept of interference sensitive WCET (isWCET), to account for shared resource contention in COTS multicore processors. isWCET is calculated offline by analyzing the WCET and tasks' resource usage in isolation and the computed shared resource interference delays from co-executing tasks. In addition, they have an online mechanism based on hardware performance monitor events to bound a task's (core's) maximum resource contention.

Few previous works adopt an existing hardware feature, Intel CAT [144], to enable the partitioning of the LLC, while some others [159, 106, 42] design bandwidth limitation feature at the hardware level. Pagani et at. [106, 125] and Aghilinasab et al. [6] discuss the integration of application cores with accelerators and GPUs in a shared memory system, respectively. Serrano-Cases et al. [118] performed qualitative and quantitative analysis on various QoS-enabled IP blocks on a Zynq UltraScale+ and provided suggestions to control contention in the shared bus and memory controller at runtime. Envelope-aWare Predictive model (E-WarP) [125] is a framework to profile tasks executing on CPU cores and accelerators. The framework bounds the execution time of tasks by monitoring the memory controller activity via performance monitor events and enforcing memory-bandwidth regulation by a combination of Memguard [151] (for CPUs) and ARM QoS Support (for accelerators).

Recently, Intel also supported the Resource Limitation approach on the new Xeon Scalable Processors at the hardware level. Intel MBA delays the requests going to the High-speed interconnect from a core's private context. Intel provides nine values between 10 to 90 in increments of 10 representative of the delays inserted between the requests. The memory bandwidth allocation/regulation is an indirect consequence of the intro-

duced delay. Similarly, Memory System Resource Partitioning and Monitoring (MPAM) [82] from ARM provide memory-access regulation mechanisms in A-profile ARM processors. Zini et al. [162] take a closer look at this technology and provide detailed instructions for MPAM instantiation. Recently, another work [124] has explored the indirect memory bandwidth limitation of the MBA. However, they adopt old synthetic benchmarks for evaluation. These benchmarks are insufficient to reproduce the worst case in the new micro-architectures. Contrarily, in our work [41], we build a new synthetic benchmark with a new traffic pattern for the exclusive cache architecture, reproducing the worst-case interference. Taking inspiration from [124], we call this type of workload "Exclusive DRAM Bomb," considering the Exclusivity of the LLC.

8.1.3 Resource Reservation

The last relevant approach is Resource Reservation. Memory banks can work in parallel; assigning distinct banks to critical and non-critical applications reduces contention and access variability. Several works [73, 154, 132, 127] consolidate this solution with concrete implementations. However, the throughput is reduced in favour of predictability. Moreover, the binding of memory banks with physical addresses is generally not directly available on COTS platforms and requires reverse engineering techniques to be operational. PALLOC [150] extends Linux virtual memory system to allocate memory pages of tasks to specific DRAM banks.

Specialized virtualization technologies (e.g., XtratuM hypervisor [89]) provide strong spatial isolation at DRAM level by allowing a system designer to allocate memory regions to specific virtual machines. However, they do not support memory bandwidth allocation to VMs.

Yun et al. extended the resource limitation method [151] to develop MemGuard [152], a memory reservation mechanism that statically partitions memory bandwidth between cores. Memguard divides the memory bandwidth between guaranteed and best effort. Once all cores have exhausted their guaranteed bandwidth, they compete with other cores for the best-effort bandwidth. Mancuso et al. [87] propose a Single Core Equivalence (SCE) framework by statically allocating an equal amount of each shared resource, such as DRAM banks, memory bandwidth, and shared cache, to each core of the COTS multicore platform. The frame-

work aims to provide a parametric WCET estimation for a task running on top of such a statically partitioned platform. Contrary to the static memory bandwidth allocation in the SCE framework, Agrawal et al. [9] consider dynamic memory bandwidth allocation and analyze the worst-case response time of a task executing in a sequence of intervals with different memory bandwidth allocations.

Behnam et al. [15] proposed a multi-resource server-based approach on the Freescale P4080 processor to provide guaranteed CPU bandwidth and memory bandwidth to tasks. For a slot-based time-triggered system executing on a COTS multicore, Agrawal et al. [7] proposed two servers per core and used the isWCET concept [101]. One server controls the processor time, and the other manages the memory bandwidth. Thus, both servers jointly control the contention between cores and the memory accesses per slot.

In DNA [51], the authors built an execution profile of tasks consisting of phases by analyzing their resource usage patterns. A runtime mechanism dynamically allocates resources to tasks based on tasks' phases. However, DNA only considers simple phase analysis and leaves considering complex task behavior in phase analysis to future work.

8.2 Hypervisor testing

At the best of our knowledge, IRIS is the first framework that enables the record and replay in hardware-assisted virtualization. Its main difference from previous studies is that it does not build the seed manually. Instead, it allows recording the VM executions to learn valid VM seeds, and it allows following the entire hardware-assisted virtualization behaviors. In this section, we discuss previous studies on hypervisor testing and record and replay approaches for security applications.

8.2.1 CPU Virtualization testing

Amit et al. [12] propose to apply the testing environment of CPU vendors to hypervisors, however, they need an intimate awareness of x86 architecture to generate comprehensive test cases. *PokeEMU* [146] generates CPU test cases for virtual CPU implementations applying symbolic

execution exclusively to an executable specification, without considering the implementation. However, its main targets are hypervisor with no hardware-assisted virtualization. Similarly, *MultiNyx* [46] generates test cases focusing on hardware-assisted virtualization, by applying dynamic symbolic execution. However, *MultiNyx* records multiple traces between VM and VMM context incurring a high performance overhead. *HyperFuzzer* [49] is a hybrid fuzzer for virtual CPUs. Both *HyperFuzzer* and *MultiNyx* are snapshot-based fuzzer. Its main difference from [46, 146] is that it avoids the overhead of a full hypervisor execution track, relying on instrumentation. Instead, it only records the program's control flow by using commodity hardware tracing. These studies construct the initial fuzzing seeds manually based on expert knowledge. In addition, they do not focus on I/O device virtualization behaviors.

8.2.2 Device Virtualization testing

The following studies do not mutate the VM's architectural state. This can limit their testing coverage, as the hypervisor depends on the VM's architectural state when emulating an operation. *Schumilio et al.* [115] first discover the available hypervisor interfaces via a custom OS, then they test such interfaces through a black-box fuzzer based on a custom bytecode interpreter which accelerates the input generation phase. Once again, the fuzzing seeds are built manually. *Nyx* [116] tests the hypervisor target via nested virtualization using KVM. In addition, *Nyx* uses grammar rules to specify the structure of the target emulated devices. Relying on manual input grammars per device requires manual work to specify grammar rules [95], thereby several studies record the interactions between the guest operating system and the device [95, 59, 107, 26]. *Henderson et al.* [59] selectively instrument the code of a given virtual device, and perform a record and replay of the only memory-mapped I/O (MMIO) activity of the virtual device in QEMU. *VShutlle*, *Morphuzz*, and *MundoFuzz* [107, 26, 95] fuzz the entire emulated device input interface including DMA interactions. Contrary to MMIO and PIO interactions that call the hypervisor intervention interrupting the VM (VM exit), the DMA does not interrupt the VM. Indeed, both the work [107, 26] instrument the DMA API of the Hypervisor to target the dynamic memory regions where the DMA is working. Instead, *MundoFuzz* [95] collects IO instructions

and DMA operations within the guest operating system without hypervisor instrumentation. MundoFuzz [95] fuzz the hypervisor with grammar awareness using automatic grammar inference. Hypervisor grammars have hidden input semantics, and *MundoFuzz* finds the causal relationships between the inputs through experiments (statistical learning). Additionally, the recorded inputs could be interleaved from asynchronous events (e.g., the timer interrupts) that generate coverage noises. *MundoFuzz* deletes this noise through differential learning.

8.2.3 Record and replay

In fuzzing, the record and replay is an effective way to learn the grammar of the target system [95, 59, 107, 120]. However, record and replay are also adopted in security to analyze and debug execution traces. Record and deterministic Replay (RnR) is a popular architectural technique [25, 34, 28, 35, 119, 134]. The RnR injects the recorded events at the correct times, enforcing a deterministic execution (Replay). RnR is used for several reasons. For instance, when the system adopts no precise events to detect possible exploits and violations, the replay is used to verify if those events are false positives [119]. It is also used to analyze time-of-check to time-of-use race conditions [35] or to determine if systems were previously exploited once zero-day attacks are discovered [69]. RnR can be done at different abstraction layers, however, to the best of our knowledge we are the first to record and replay the VMM history in hardware-assisted virtualization solutions.

Chapter 9

Conclusion

This thesis contributed to two key limitations of the emerging real-time cloud paradigm: memory access isolation and failure isolation across execution environments hosted on the same hardware platform.

In terms of memory access isolation, we provided a systematic approach to evaluate the actual regulation of black-box hardware controller specialized for resource limitation, and we propose memory queue occupancy as an observable metric to estimate the current degree of memory access interference, useful for applying regulation only in case of interference, improving memory bandwidth utilization. From our experiments, we observed that the synthetic workloads we designed for the evaluation of the Intel Memory Bandwidth Allocation can bypass over 50% the regulation shown by related works, and the memory queue occupancy can detect the degree of interference within one-fifth of the regulation period.

Regarding failure isolation, we provided a new record and replay method to test hypervisor robustness, the most privileged and shared software layer on cloud platforms. We implemented IRIS, the first intrusive prototype of this method in Xen, which requires instrumenting and modifying the Xen code base. Our preliminary experiments showed that IRIS can automatically reach valid hypervisor states, with a significant time improvement (from 42.5% to 99,6%) and with high accuracy (with code coverage fitting between 92.1% and 100%). This enables the application of mutations in specific states to generate new test cases without manual intervention. We further introduced HyRo, an extension of IRIS designed

to operate transparently to the target, eliminating the need for code modifications. In the next section, we discuss the limitations and future directions of this thesis.

9.1 Memory Access Isolation

According to Mind Commerce (2020), over 70% of IoT applications in industrial automation will depend on real-time operating systems (RTOS) by 2025. However, Industrial Internet of Things (IIoT) deployments require both high performance and predictability to support the high connectivity and integration demands of OT (Operational Technology) and IT (Information Technology) systems. OT includes control and monitoring features within industrial environments, while IT refers to information management infrastructure. This dual demand is pushing hardware vendors like Intel to enhance spatial and temporal isolation capabilities in their processors, critical for meeting IIoT application requirements.

MBA evaluation. In this thesis, we evaluated MBA1.0 (Memory Bandwidth Allocation), especially its ability to safeguard critical applications against performance degradation caused by memory access contention. Future work will extend this evaluation to the newer MBA versions, focusing even on security vulnerabilities, such as contention-based side channels, which represent a growing risk in multi-tenant environments.

New hardware features. Additionally, several emerging hardware features were outside this thesis's scope but require systematic evaluation and support, including Time Coordinated Computing (TCC), time-sensitive networking (TSN), and functional safety protocols, especially because some of these features are already active on the field [140] in industrial automation and robotics. We aim to validate these platforms and hardware features in future work, to build a fully-flagged real-time cloud environment for IIoT.

9.2 Hypervisor Robustness

Despite positive results obtained by using the proposed record and replay framework, we briefly discuss limitations and avenues of further

improvement.

Replay Accuracy. Hyro framework does not replay accurately some *hypervisor behaviors* as discussed in Section 7. Specifically, the recording mechanism deliberately does not store guest VM memory areas touched during VM exit handling. We plan to explore a way to both record and replay efficiently these accesses of the hypervisor to the guest VM memory areas during hypervisor intervention.

Code coverage. The code coverage is a paramount metric to guide a fuzzer in interesting points of target source code. Our implementations at most leverage a software-based approach like *gcov* [131] to retrieve such code coverage data independently of the CPU architecture. Other hardware-based mechanisms, like *Intel Processor Trace* (Intel PT) [66], allow recording complete control flow with low-performance overhead while not modifying the target hypervisor. We plan to experiment *Intel PT* (see Chap. 35 in [66]) in HyRo to make feasible an efficient coverage-guided fuzzer. However, Intel PT can not be used for a hypervisor that does not target Intel VT-x extensions.

Fuzzing. In this study, we only provided a proof-of-concept fuzzer based upon a record/replay framework, as an example of an assessment solution that addresses hardware-assisted virtualization. However, the simpler mutation rules adopted do not cover the complex fuzzing logic that is adopted by current state-of-the-art fuzzers. We plan to perform a thorough fuzzing experimentation, exploiting the findings provided in this study, to develop a fuzzer aimed at discovering vulnerabilities for hardware-assisted hypervisors.

Bibliography

[1] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. Intel Virtualization Technology for Directed I/O. *Intel technology journal*, 10(3), 2006.

[2] Onur Aciiçmez. Yet another microarchitectural attack: exploiting i-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 11–18, 2007.

[3] Onur Acıiçmez and Werner Schindler. A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl. In *Topics in Cryptology–CT-RSA 2008: The Cryptographers' Track at the RSA Conference 2008, San Francisco, CA, USA, April 8-11, 2008. Proceedings*, pages 256–273. Springer, 2008.

[4] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices*, 41(11):2–13, 2006.

[5] Homa Aghilinasab, Waqar Ali, Heechul Yun, and Rodolfo Pellizzoni. Dynamic memory bandwidth allocation for real-time gpu-based soc platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3348–3360, 2020.

[6] Homa Aghilinasab, Waqar Ali, Heechul Yun, and Rodolfo Pellizzoni. Dynamic memory bandwidth allocation for real-time gpu-based soc platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3348–3360, 2020.

[7] Ankit Agrawal, Gerhard Fohler, Johannes Freitag, Jan Nowotsch, Sascha Uhrig, and Michael Paulitsch. Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case

Study. In Marko Bertogna, editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:22, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[8] Ankit Agrawal, Gerhard Fohler, Jan Nowotsch, Sascha Uhrig, and Michael Paulitsch. Poster abstract: Slot-level time-triggered scheduling on cots multicore platform with resource contentions. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[9] Ankit Agrawal, Renato Mancuso, Rodolfo Pellizzoni, and Gerhard Fohler. Analysis of dynamic memory bandwidth regulation in multi-core real-time systems, 2018.

[10] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 870–887. IEEE, 2019.

[11] AMD Technology. AMD64 Architecture Programmer's Manual Volume 2: System Programming.

[12] Nadav Amit, Dan Tsafrir, Assaf Schuster, Ahmad Ayoub, and Eran Shlomo. Virtual CPU validation. *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

[13] Étienne André, Remi Dulong, Amina Guermouche, and François Trahay. duf: Dynamic uncore frequency scaling to reduce power consumption. *Concurrency and Computation: Practice and Experience*, 34, 2022.

[14] Lucian Armasu. Openbsd will disable intel hyper-threading to avoid spectre-like exploits (updated). 2018.

[15] Moris Behnam, Rafia Inam, Thomas Nolte, and Mikael Sjödin. Multi-core composability in the face of memory-bus contention. *SIGBED Rev.*, 10(3), October 2013.

[16] Frank Bellosa. Process cruise control: Throttling memory access in a soft real-time environment. In *SOSP 1997*, 1997.

[17] Frank Bellosa. The third dimension of scheduling. 1999.

[18] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The turtles project: Design and implementation of nested virtualization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 423–436, USA, 2010. USENIX Association.

[19] Daniel J Bernstein. Cache-timing attacks on aes. 2005.

[20] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 785–800, 2019.

[21] Alessandro Biondi and Marco Di Natale. Achieving predictable multicore execution of automotive applications using the let paradigm. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 240–250, 2018.

[22] F. Boniol, H. Cassé, É. Noulard, and C. Pagetti. Deterministic execution model on cots hardware. In *ARCS*, 2012.

[23] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against aes. In *Cryptographic Hardware and Embedded Systems-CHES 2006: 8th International Workshop, Yokohama, Japan, October 10-13, 2006. Proceedings 8*, pages 201–215. Springer, 2006.

[24] Michael Brengel, Michael Backes, and Christian Rossow. Detecting hardware-assisted virtualization. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*, pages 207–227. Springer, 2016.

[25] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. In *TOCS*, 1996.

[26] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. Morphuzz: Bending (Input) Space to Fuzz Virtual Devices. In *USENIX Security Symposium*, 2022.

[27] Sudipta Chattopadhyay, Lee Kee Chong, Abhik Roychoudhury, Timon Kelter, P. Marwedel, and H. Falk. A unified wcet analysis framework for multi-core platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2012.

[28] Jim Chow, Tal Garfinkel, and Peter M. Chen. VMwareDecoupling Dynamic Program Analysis from Execution in Virtual Environments. In *USENIX Annual Technical Conference*, 2008.

[29] Marcello Cinque, Domenico Cotroneo, Luigi De Simone, and Stefano Rosiello. Virtualizing mixed-criticality systems: A survey on industrial trends and issues. *Future Generation Computer Systems*, 129:315–330, 2022.

[30] Domenico Cotroneo, Luigi De Simone, and Roberto Natella. On temporal isolation assessment in virtualized railway signaling as a service systems, 2022.

[31] Miles Dai, Riccardo Paccagnella, Miguel Gomez-Garcia, John McCalpin, and Mengjia Yan. Don't mesh around:{Side-Channel} attacks and mitigations on mesh interconnects. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2857–2874, 2022.

[32] C. Dall, S. Li, J. T. Lim, J. Nieh, and G. Koloventzos. ARM virtualization: performance and architectural implications. In *Annual International Symposium on Computer Architecture*, pages 304–316. IEEE, 2016.

[33] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. {Prime+ Abort}: A {Timer-Free}{High-Precision} l3 cache attack using intel {TSX}. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 51–67, 2017.

[34] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *OPSR*, 2002.

[35] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *VEE '08*, 2008.

[36] Stephen A. Edwards and Edward A. Lee. The case for the precision timed (pret) machine. In *2007 44th ACM/IEEE Design Automation Conference*, pages 264–265, 2007.

[37] Ericsson. *Network Compute Fabric*. Last accessed:08/21.

[38] Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 843–857, 2016.

[39] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[40] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Understanding and mitigating covert channels through branch predictors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):1–23, 2016.

[41] Giorgio Farina, Gautam Gala, Marcello Cinque, and Gerhard Fohler. Assessing intel's memory bandwidth allocation for resource limitation in real-time systems. 2022.

[42] Farzad Farshchi, Qijing Huang, and Heechul Yun. Bru: Bandwidth regulation unit for real-time multicore processors. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 364–375, 2020.

[43] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, and Dejan Kostic. Make the most out of last level cache in intel processors. *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019.

[44] L Fenn, Richard D Hawkins, PJ Williams, Tim P Kelly, Michael G Banner, and Y Oakshott. The who, where, how, why and when of modular and incremental certification. In *2007 2nd Institution of Engineering and Technology International Conference on System Safety*, pages 135–140. IET, 2007.

[45] Gerhard Fohler, Gautam Gala, Daniel Gracia Pérez, and Claire Pagetti. Evaluation of DREAMS resource management solutions on a mixed-critical demonstrator. In *ERTS 2018*, 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018), Toulouse, France, January 2018.

[46] Pedro Fonseca, Xi Wang, and Arvind Krishnamurthy. MultiNyx: a multi-level abstraction framework for systematic analysis of hypervisors. *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[47] Gautam Gala, Gerhard Fohler, Peter Tummeltshammer, Stefan Resch, and Reingard Hametner. RT-cloud: Virtualization technologies and cloud computing for railway use-case. In *24th IEEE International Symposium On Real-Time distributed Computing (IEEE ISORC)*. IEEE, 2021.

[48] Gautam Jayantilal Gala. *Resource Management for Real-time and Mixed-Critical Systems*. PhD thesis, Technische Universität Kaiserslautern, 2021.

[49] Xinyang Ge, Ben Niu, Robert Brotzman, Yaohui Chen, HyungSeok Han, Patrice Godefroid, and Weidong Cui. HyperFuzzer: An Efficient Hybrid Fuzzer for Virtual CPUs. *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[50] Daniel Genkin, Luke Valenta, and Yuval Yarom. May the fourth be with you: A microarchitectural side channel attack on several real-world applications of curve25519. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 845–858, 2017.

[51] Robert Gifford, Neeraj Gandhi, Linh Thi Xuan Phan, and Andreas Haeberlen. Dna: Dynamic resource allocation for soft real-time multicore systems. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 196–209, 2021.

[52] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In *NDSS*, 2020.

[53] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 955–972, 2018.

[54] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*, pages 279–299. Springer, 2016.

[55] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive {Last-Level} caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, 2015.

[56] Reinhard Hametner, Peter Tummeltshammer, Stefan Resch, and Wolfgang Wernhar. Cloud architecture for sil4 railway applications. *Thales Austria GmbH, Signal + Draht - SIGNALLING & DATACOMMUNICATION magazine, Eurailpress Archiv - DVV Media Group GmbH*, March 2022.

[57] Yacine Hebbal, Sylvie Laniepce, and Jean-Marc Menaud. Virtual Machine Introspection: Techniques and Applications. *2015 10th International Conference on Availability, Reliability and Security*, pages 676–685, 2015.

[58] Christian Helm, Soramichi Akiyama, and Kenjiro Taura. Reliable reverse engineering of intel dram addressing using performance counters. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8, 2020.

[59] Andrew Henderson, Heng Yin, Guang yao Jin, Hao Han, and Hongmei Deng. VDF: Targeted Evolutionary Fuzz Testing of Virtual Devices. In *RAID*, 2017.

[60] Přemysl Houdek, Michal Sojka, and Zdeněk Hanzálek. Towards predictable execution model on arm-based heterogeneous platforms. In *2017 IEEE 26th International Symposium on Industrial Electronics (ISIE)*, pages 1297–1302, 2017.

[61] Jingyuan Hu, Xiaokuang Bai, Sai Sha, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. Hub: Hugepage ballooning in kernel-based virtual machines. In *Proceedings of the International Symposium on Memory Systems*, pages 31–37, 2018.

[62] Intel. IntelResource Director Technology (Intel RDT) On 2nd Generation Intel Xeon Scalable Processor Reference Manual, April 2019.

[63] Intel. Intel Architecture Instruction Set Extensions and Future Features Programming Reference, January 2018.

[64] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. November 2020.

[65] Intel. *Intel Xeon Processor Scalable Memory Family Uncore Performance Monitoring*. November 2020.

[66] CO Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide. *Denver,[2006]*, 2022.

[67] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. A shared cache attack that works across cores and defies vm sandboxing–and its application to aes. In *2015 IEEE Symposium on Security and Privacy*, pages 591–604. IEEE, 2015.

[68] ISO. Road vehicles – Functional safety, 2011.

[69] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Symposium on Operating Systems Principles*, 2005.

[70] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[71] Timon Kelter, Tim Harde, Peter Marwedel, and Heiko Falk. Evaluation of resource arbitration methods for multi-core real-time systems. In *13th International Workshop on Worst-Case Execution Time Analysis*, Dagstuhl, Germany, 2013.

[72] Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154. IEEE, 2014.

[73] Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154, 2014.

[74] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. {STEALTHMEM}:{System-Level} protection against {Cache-Based} side channel attacks in the cloud. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 189–204, 2012.

[75] Samuel T King and Peter M Chen. Subvirt: Implementing malware with virtual machines. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 14–pp. IEEE, 2006.

[76] Christoph M. Kirsch and Ana Sokolova. *The Logical Execution Time Paradigm*. Springer Berlin Heidelberg, 2012.

[77] NG Chetan Kumar, Sudhanshu Vyas, Ron K Cytron, Christopher D Gill, Joseph Zambreno, and Phillip H Jones. Cache design for mixed criticality real-time systems. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 513–516. IEEE, 2014.

[78] Oren Laadan and Jason Nieh. Operating system virtualization: practice and experience. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, pages 1–12, 2010.

[79] Marianna Lezzi, Mariangela Lazoi, and Angelo Corallo. Cybersecurity for Industry 4.0 in the current literature: A reference framework. *Computers in Industry*, 103:97–110, 2018.

[80] Zhuozhao Li, Tanmoy Sen, Haiying Shen, and Mooi Choo Chuah. A study on the impact of memory dos attacks on cloud applications and exploring real-time detection schemes. *IEEE/ACM Transactions on Networking*, 30(4):1644–1658, 2022.

[81] Min Yeol Lim, Allan Porterfield, and Robert J. Fowler. Softpower: fine-grain power estimations using performance counters. In *HPDC '10*, 2010.

[82] Arm Limited. *Arm Architecture Reference Manual Supplement Memory System Resource Partitioning and Monitoring (MPAM), for A-profile architecture*. 2022. ARM DDI 0598D.b (ID073122).

[83] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a way: Exploring the security implications of amd's cache way predictors. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 813–825, 2020.

[84] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418, 2016.

[85] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.

[86] Mailing list. Xen Memory De-duplication.

[87] Renato Mancuso, Rodolfo Pellizzoni, Marco Caccamo, Lui Sha, and Heechul Yun. Wcet(m) estimation in multi-core systems using single core equivalence. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 174–183, 2015.

[88] Renato Mancuso, Heechul Yun, and Isabelle Puaut. Impact of dm-lru on wcet: a static analysis approach. In *ECRTS 2019-31st Euromicro Conference on Real-Time Systems*, pages 1–25, 2019.

[89] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and J Metge. Xtratum: a hypervisor for safety critical embedded systems. In *11th Real-Time Linux Workshop*, pages 263–272. Citeseer, 2009.

[90] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: cross-cores cache covert channel. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings 12*, pages 46–64. Springer, 2015.

[91] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In *RAID*, 2015.

[92] Daniel Molka, Robert Schöne, Daniel Hackenberg, and Wolfgang E. Nagel. Detecting memory-boundedness with hardware performance counters. *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 2017.

[93] ABM Moniruzzaman. Analysis of memory ballooning technique for dynamic memory management of virtual machines (vms). *International Journal of Grid and Distributed Computing*, 7(6):81–90, 2014.

[94] Thomas Moscibroda Onur Mutlu. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX security*, 2007.

[95] Cheolwoo Myung, Gwangmu Lee, and Byoungyoung Lee. MundoFuzz: Hypervisor Fuzzing with Statistical Coverage Testing and Grammar Inference. In *USENIX Security Symposium*, 2022.

[96] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(3), 2006.

[97] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3), 2006.

[98] NIST. CVE-2010-0435.

[99] NIST. CVE-2011-1936.

[100] NIST. CVE-2020-2732.

[101] Jan Nowotsch, Michael Paulitsch, Daniel Bühler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 109–118. IEEE, 2014.

[102] Online. *Digitale Schiene Deutschland*. Last accessed:12/22.

[103] Online. *Scalable Open Architecture for Embedded Edge (SOAFEE)*. Last accessed:12/22.

[104] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology–CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*, pages 1–20. Springer, 2006.

[105] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. Lord of the ring (s): Side channel attacks on the {CPU}{On-Chip} ring interconnect are practical. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 645–662, 2021.

[106] Marco Pagani, Enrico Rossi, Alessandro Biondi, Mauro Marinoni, Giuseppe Lipari, and Giorgio C. Buttazzo. A bandwidth reservation mechanism for axi-based hardware accelerators on fpgas. In *ECRTS*, 2019.

[107] Gaoning Pan, Xingwei Lin, Xuhong Zhang, Yongkang Jia, Shouling Ji, Chunming Wu, Xinlei Ying, Jiashui Wang, and Yanjun Wu. V-Shuttle: Scalable and Semantics-Aware Hypervisor Virtual Device Fuzzing. *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[108] R. Pellizzoni, E. Betti, Stanley Bak, Gang Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded systems. *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, 2011.

[109] R. Pellizzoni, A. Schranzhofer, Jian-Jia Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 741–746, 2010.

[110] Colin Percival. Cache missing for fun and profit, 2005.

[111] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. In *CACM*, 1973.

[112] Jan Reineke, Isaac Liu, Hiren D. Patel, Sungjun Kim, and Edward A. Lee. Pret dram controller: Bank privatization for predictability and temporal isolation. In *2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 99–108, 2011.

[113] RTCA. DO-178C Software Considerations in Airborne Systems and Equipment Certification *Requirements and Technical Concepts for Aviation*, 1992.

[114] Andreas Schranzhofer, Rodolfo Pellizzoni, Jian-Jia Chen, Lothar Thiele, and Marco Caccamo. Timing analysis for resource access interference on adaptive resource arbiters. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 213–222, 2011.

[115] Sergej Schumilo, Cornelius Aschermann, Ali Reza Abbasi, Simon Wörner, and Thorsten Holz. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In *NDSS*, 2020.

[116] Sergej Schumilo, Cornelius Aschermann, Ali Reza Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *USENIX Security Symposium*, 2021.

[117] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clementine Lucie Noemie Maurice, Raphael Spreitzer, and Stefan Mangard. Keydrown: Eliminating software-based keystroke timing side-channel attacks. In *Network and Distributed System Security Symposium 2018*, page 15, 2018.

[118] Alejandro Serrano-Cases, Juan M. Reina, Jaume Abella, Enrico Mezzetti, and Francisco J. Cazorla. Leveraging Hardware QoS to Control Contention in the Xilinx Zynq UltraScale+ MPSoC. In Björn B. Brandenburg, editor,

33rd Euromicro Conference on Real-Time Systems (ECRTS 2021), volume 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:26, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[119] Yasser Shalabi, Mengjia Yan, Nima Honarmand, Ruby B. Lee, and Josep Torrellas. Record-Replay Architecture as a General Security Framework. *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 180–193, 2018.

[120] Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt. Drifuzz: Harvesting Bugs in Device Drivers from Golden Seeds. In *USENIX Security Symposium*, 2022.

[121] Girish Shirasat and ARM. *The cloud-native approach to the software defined car*. Last accessed:12/22.

[122] Siemens. *Infrastructure in the Cloud*. Last accessed:12/22.

[123] Siemens AG. Jailhouse hypervisor source code.

[124] Parul Sohal, Michael Garrett Bechtel, Renato Mancuso, Heechul Yun, and Orran Krieger. A closer look at intel resource director technology (rdt). *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, 2022.

[125] Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. E-warp: A system-wide framework for memory bandwidth profiling and management. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 345–357, 2020.

[126] Read Sprabery, Konstantin Evchenko, Abhilash Raj, Rakesh B Bobba, Sibin Mohan, and Roy Campbell. Scheduling, isolation, and cache allocation: A side-channel defense. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 34–40. IEEE, 2018.

[127] Noriaki Suzuki, Hyoseung Kim, Dionisio de Niz, Bjorn Andersson, Lutz Wrage, Mark Klein, and Ragunathan Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *2013 IEEE 16th International Conference on Computational Science and Engineering*, pages 685–692, 2013.

[128] Thales. *TransVital*. Last accessed:12/22.

[129] Thales, SYSGO, Fraunhofer IESE, University of Rostock, ESE, and DB Netz. *Research Report - SIL4 Cloud*. Last accessed:12/22.

[130] The Linux Foundation. Clock sources, Clock events, sched_clock() and delay timers.

[131] The Linux Foundation. Gcov Home page.

[132] Prathap Kumar Valsan and Heechul Yun. Medusa: A predictable and high-performance dram controller for multicore based embedded systems. In *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications*, pages 86–93, 2015.

[133] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. Meshup: Stateless cache side-channel attack on cpu mesh. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1506–1524. IEEE, 2022.

[134] Wei Wang, Zhiyu Hao, and Lei Cui. ClusterRR: a record and replay framework for virtual machine cluster. *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2022.

[135] Yao Wang, Andrew Ferraiuolo, and G Edward Suh. Timing channel protection for a shared memory controller. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 225–236. IEEE, 2014.

[136] Zhenghong Wang and Ruby B Lee. Covert and side channels due to processor architecture. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 473–482. IEEE, 2006.

[137] Bryan C. Ward, Jonathan L. Herman, Christopher J. Kenna, and James H. Anderson. Outstanding paper award: Making shared caches more predictable on multicore platforms. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 157–167, 2013.

[138] Song Wei, Kun Zhang, and Bibo Tu. Hyperbench: A benchmark suite for virtualization capabilities. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):1–22, 2019.

[139] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves. In *Computer Security–ESORICS 2016: 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I 21*, pages 440–457. Springer, 2016.

[140] Wind River Systems, Inc. Wind River and Intel: Real-Time and Functional Safety Solutions on Intel Industrial Platforms, 2021.

[141] Xen project. Hvmloader.

[142] Sisu Xi, Meng Xu, Chenyang Lu, Linh TX Phan, Christopher Gill, Oleg Sokolsky, and Insup Lee. Real-time multi-core virtual machine scheduling in Xen. In *Proc. EMSOFT*, pages 1–10. IEEE, 2014.

[143] Yaocheng Xiang, Chencheng Ye, Xiaolin Wang, Yingwei Luo, and Zhenlin Wang. Emba: Efficient memory bandwidth allocation to improve performance on intel commodity processor. *Proceedings of the 48th International Conference on Parallel Processing*, 2019.

[144] Meng Xu, Linh Thi Xuan Phan, Hyon-Young Choi, Yuhan Lin, Haoran Li, Chenyang Lu, and Insup Lee. Holistic resource allocation for multicore real-time systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 345–356, 2019.

[145] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 888–904. IEEE, 2019.

[146] Qiuchen Yan and Stephen McCamant. Fast PokeEMU: Scaling Generated Instruction Tests Using Aggregation and State Chaining. *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2018.

[147] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48(6):681–715, 2012.

[148] Yuval Yarom and Katrina Falkner. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *23rd USENIX security symposium (USENIX security 14)*, pages 719–732, 2014.

[149] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7:99–112, 2017.

[150] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, 2014.

[151] H. Yun, Gang Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. *2012 24th Euromicro Conference on Real-Time Systems*, pages 299–308, 2012.

[152] H. Yun, Gang Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.

[153] Heechul Yun, Waqar Ali, Santosh Gondi, and Siddhartha Biswas. Bwlock: A dynamic memory access control framework for soft real-time applications on multicore platforms. *IEEE Transactions on Computers*, 66(7):1247–1252, 2016.

[154] Heechul Yun, Rodolfo Pellizzon, and Prathap Kumar Valsan. Parallelism-aware memory interference delay analysis for cots multicore systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 184–195, 2015.

[155] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Dos attacks on your memory in cloud. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 253–265, 2017.

[156] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 305–316, 2012.

[157] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 990–1003, 2014.

[158] Wu Zhenyu, Xu Zhang, and H Wang. Whispers in the hyper-space: high-speed covert channel attacks in the cloud. In *USENIX Security symposium*, pages 159–173, 2012.

[159] Yanqi Zhou and David Wentzlaff. Mitts: Memory inter-arrival time traffic shaping. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 532–544, 2016.

[160] Ziqiao Zhou, Michael K Reiter, and Yinqian Zhang. A software approach to defeating side channels in last-level caches. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 871–882, 2016.

[161] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.

[162] Matteo Zini, Daniel Casini, and Alessandro Biondi. Analyzing arm's mpam from the perspective of time predictability. *IEEE Transactions on Computers*, pages 1–14, 2022.

Author's publications

1. Giorgio Farina, Gautam Gala, Marcello Cinque, Gerhard Fohler. Enabling memory access isolation in real-time cloud systems using Intel's detection/regulation capabilities. International Journal of Systems Architecture, Volume 137, April 2023, 102848, DOI: 10.1016/j.sysarc.2023.102848

2. Giorgio Farina, Gautam Gala, Marcello Cinque, Gerhard Fohler. Assessing Intel's Memory Bandwidth Allocation for resource limitation in real-time systems. IEEE 25th International Symposium On Real-Time Distributed Computing (ISORC). Västerås, Sweden, 17-18 May 2022, IEEE, DOI: 10.1109/ISORC52572.2022.9812757

3. Carmine Cesarano; Marcello Cinque; Domenico Cotroneo; Luigi De Simone; Giorgio Farina. IRIS: a Record and Replay Framework to Enable Hardware-assisted Virtualization Fuzzing. 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Porto, Portugal, 27-30 June 2023, IEEE, DOI: 10.1109/DSN58367.2023.00045.

4. Marco Barletta, Marcello Cinque, Luigi De Simone, Raffaele Della Corte, Giorgio Farina, Daniele Ottaviano. RunPHI: Enabling Mixed-criticality Containers via Partitioning Hypervisors in Industry 4.0. IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Charlotte, NC, USA, p. 134-135, 26 December 2022, IEEE, DOI: 10.1109/ISSREW55968.2022.00058

5. Marco Barletta, Marcello Cinque, Luigi De Simone, Raffaele Della Corte, Giorgio Farina, Daniele Ottaviano. Partitioned Containers: Towards Safe Clouds for Industrial Applications. 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S), Porto, Portugal, p. 84-88, June 2023, IEEE, DOI: 10.1109/DSN-S58398.2023.00029

6. Marcello Cinque, Raffaele Della Corte, Giorgio Farina, Stefano Rosiello. An unsupervised approach to discover filtering rules from diagnostic logs. IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Charlotte, NC, USA, p. 1-6, 26 December 2022, IEEE, DOI: 10.1109/ISSREW55968.2022.00030

7. Marcello Cinque, Raffaele Della Corte, Giorgio Farina, Stefano Rosiello. AID4TRAIN: Artificial Intelligence-Based Diagnostics for TRAins and INdustry 4.0. European Dependable Computing Conference 2022 Workshops, Zaragoza, Spain, vol 1656, p. 1-6, 05 September 2022, Springer, Cham, DOI: 10.1007/978-3-031-16245-9_7