Out of Hypervisor (OoH): When Nested Virtualization Becomes Practical

Stella Bitchebe

PhD Thesis Defense
February 3rd, 2023

UNIVERSITÉ CÔTE D'AZUR

Reviewers

Pr Edouard Bugnion, EPFL
Pr Gaël Thomas, Telecom SudParis

Examiners

Pr Guillaume Urvoy-Keller, UCA
Pr Laurent Réveillère, Univ. Bordeaux
Dr Natacha Crooks, UC Berkeley
Dr Oana Balmau, McGill Univ.
Dr Renaud Lachaize, Univ. Grenoble

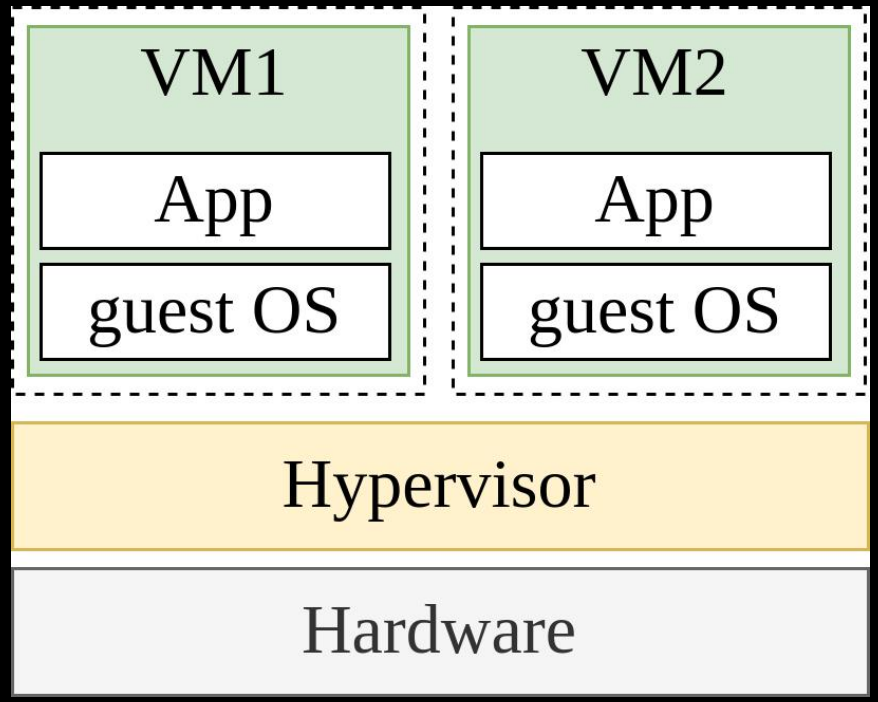
Invited

Dr Anne-Marie Chana, ENSPY
Dr Bernabe Batchakui, ENSPY

Advisor

Pr Alain Tchana, Univ. Grenoble

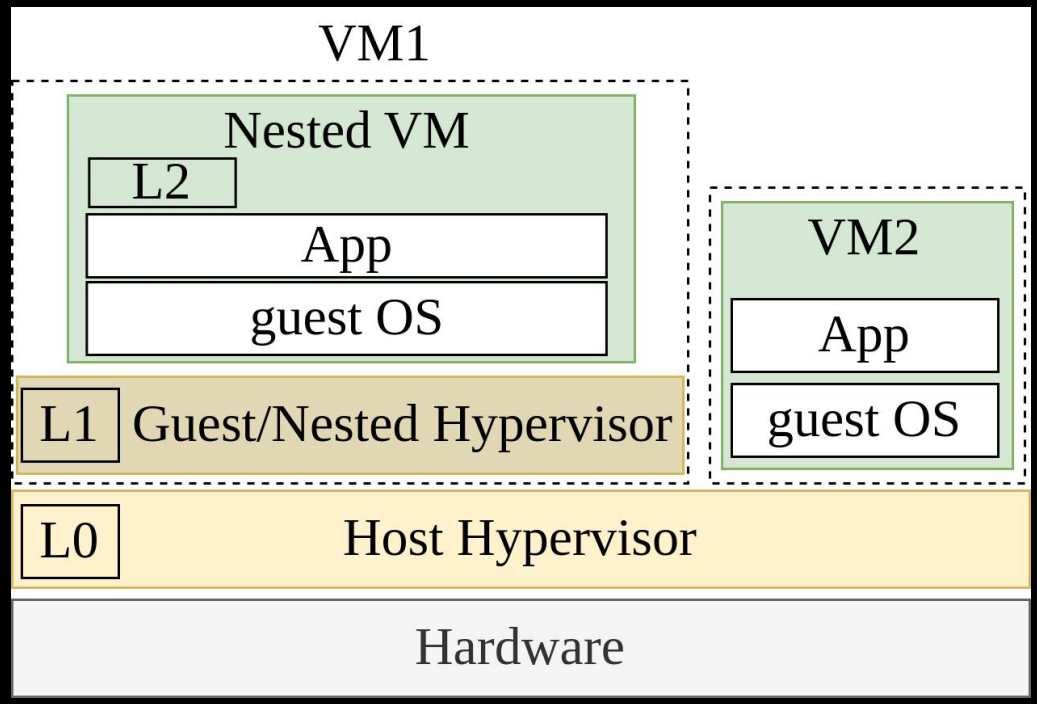
Definition



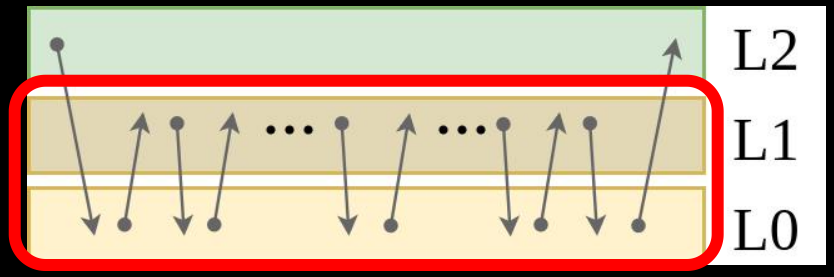
Definition

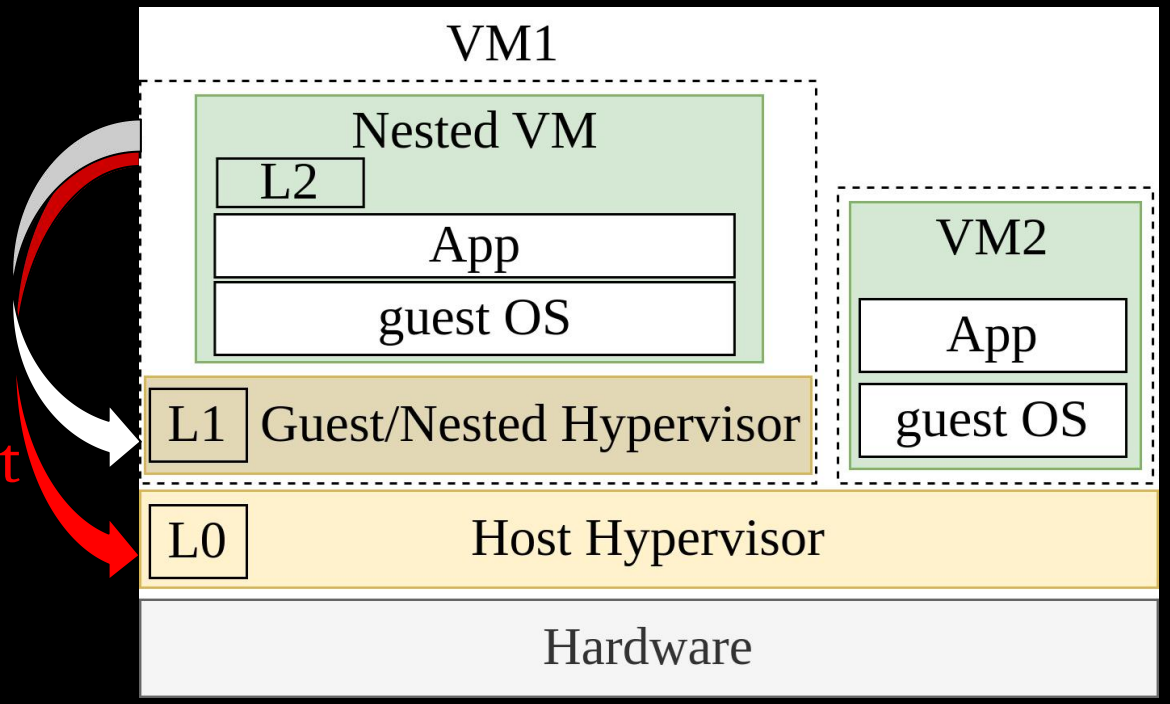
"Recursive virtualization, where a virtual machine runs under itself a copy of a hypervisor."

Popek and Goldberg, 1970s



⚠ Drawbacks





Single trap from nested VM produces multiple traps to the host hypervisor

Significant number of VM exits (at least ×2)

Vilanova et al. [150, ISCA'19]

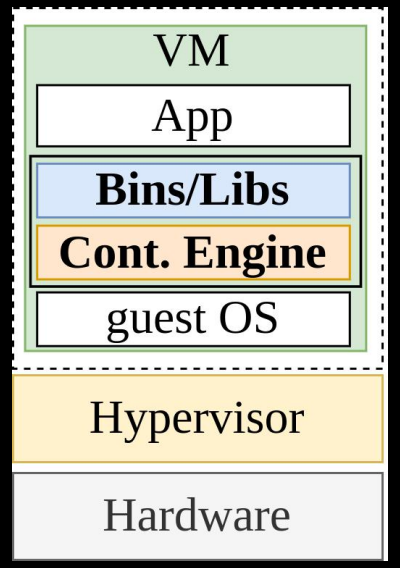
Use Cases of Nested Virt.

Tests/Development/Demo

- No matter the overhead
- Not in production

Isolation

Containers

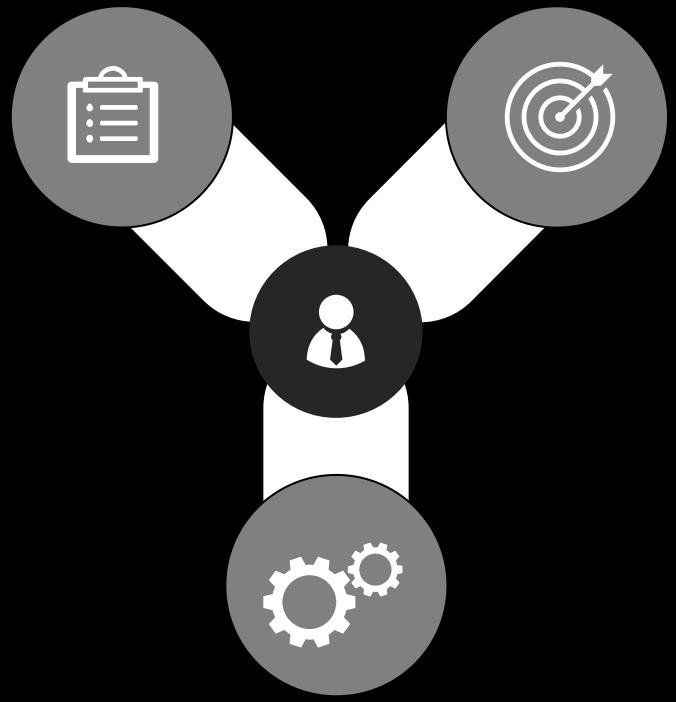


OoH: Out of Hypervisor

Presentation

What?

New virtualization research axis



Why?

Avoid trying to virtualize full virtual hardware inside a VM

How?

Expose individually hypervisor oriented hardware virt. features to the guest OS so that its processes can benefit from them

Virtualization Technologies Categorization

G1: Resource Multiplexing

- ✓ EPT (*Extended Page Table*): MMU virtualization
- ✓ SRIOV (*Single Root I/O Virtualization*)
- ✓ APICv (*Advanced Programmable Interrupt Controller virtualization*)
- ✓ etc.

G2: VM's Management

- ✓ PML (*Page Modifcation Logging*)
- ✓ CAT (*Cache Allocation Technology*)
- ✓ SPP (*Sub-Page write Permissions*)
- ✓ etc.

Virtualization Technologies Categorization

G1: Resource Multiplexing

- ✓ EPT (*Extended Page Table*): MMU virtualization
- ✓ SRIOV (*Single Root I/O Virtualization*)
- ✓ APICv (*Advanced Programmable Interrupt Controller virtualization*)
- ✓ etc.

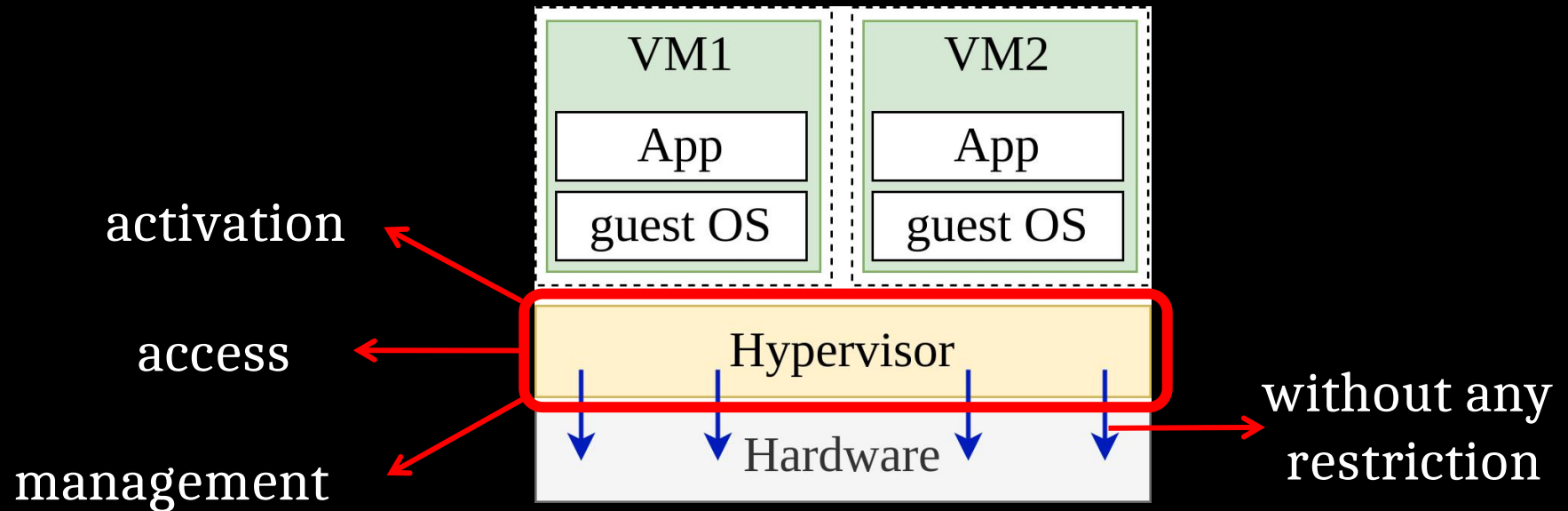
G2: VM's Management

- ✓ PML (*Page Modifcation Logging*)
- ✓ CAT (*Cache Allocation Technology*)
- ✓ SPP (*Sub-Page write Permissions*)
- ✓ etc.

OoH scope: Can be exploited to remove the need for a nested hypervisor to leverage hypervisor properties

Problem Statement

All hardware functionalities are hypervisor-oriented



State-of-the-art

Leveraging HAV: Dune

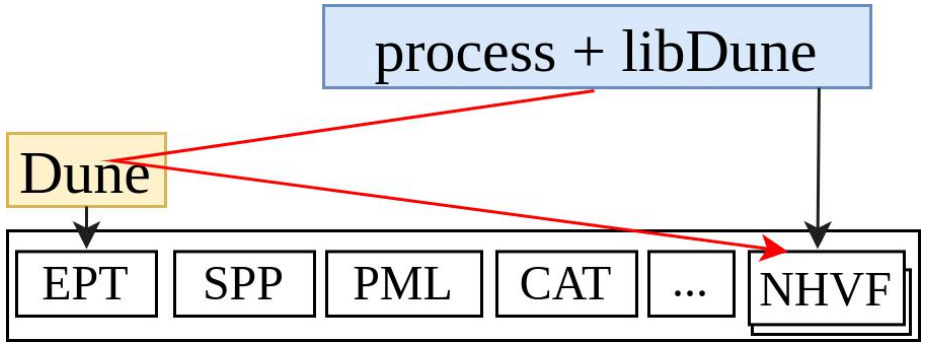
Direct Access to HW: Hardened Hypervisors

Improving Nested Virt.

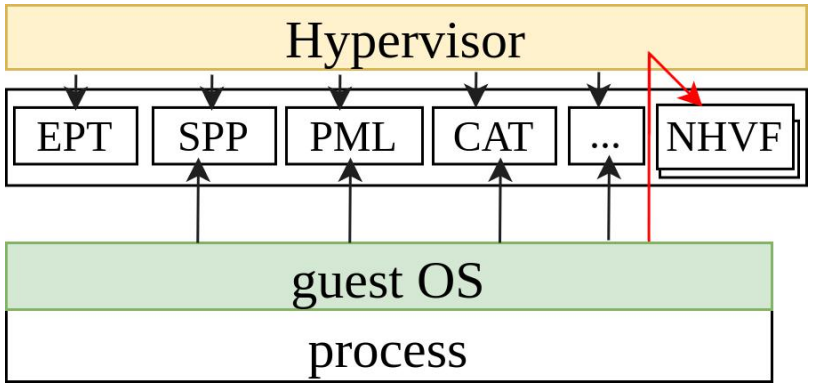
State-of-the-art

Leveraging HAV: Dune (OSDI 2012)

Dune



OoH



Dune uses hardware virt. features (EPT) to allow processes in bare-metal to access non-virt. feat. (execute privileged instructions) while OoH makes processes in VMs to access and use hardware virt. features

State-of-the-art

Direct Access to HW: Hardened Hypervisors

Nohype (ISCA 2010):
1CPU core per VM, fixed memory per VM, etc.

AWS Nitro (Amazon 2017):
network, storage, and security management from the hypv. to the HW (-> Nitro cards) to enable near-native perf

↓

Restricted #instances
Rigid hypervisor

State-of-the-art

Improving Nested Virt.

NEVE (Nested Virtualization Extensions for ARM, SOSP 2017, Lim et al.): access to VM data structures to reduce exits multiplication from nested hypv. → limited fields accessible

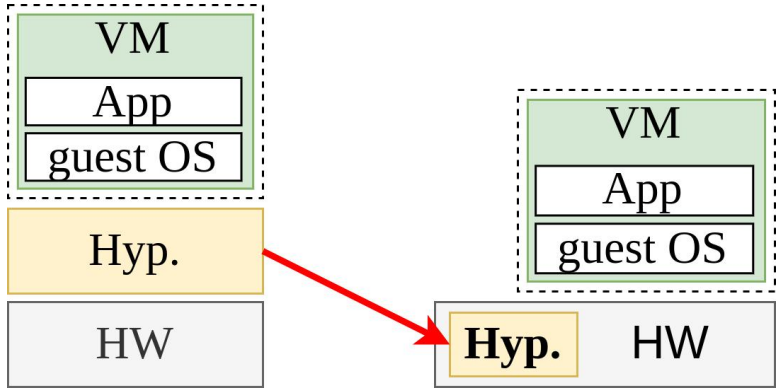
SV_T (SMT-based VirTualization, ISCA 2019, Vilonava et al.): pins each virt. layer to a single core → limits resources sharing

DVH (Direct Virtual Hardware, ASPLOS 2020, Lim et al.): direct IO devices and timers to VMs → unpractical application to all devices

OoH's State-of-the-art

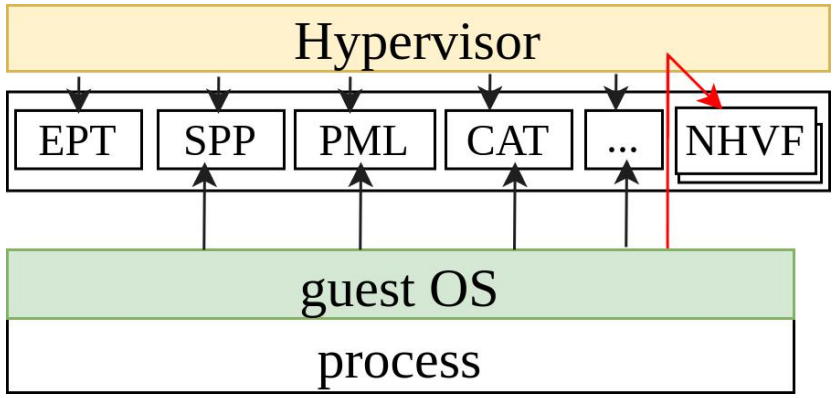
OoH in the virtualization landscape

Hardened Hyp.



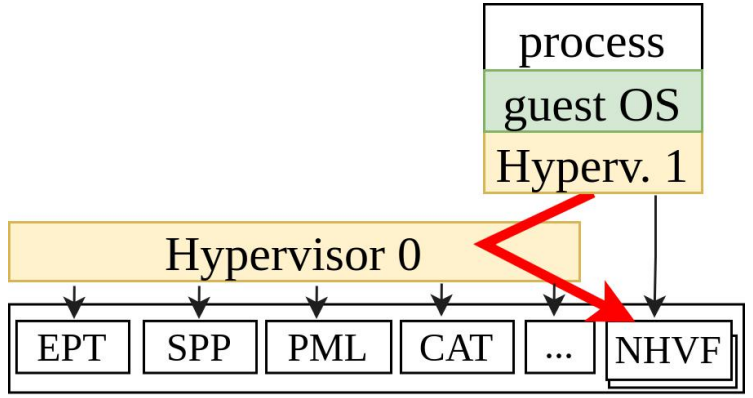
- ✓ No nested hyp.
- ✗ Rigid hypervisor
- ✗ Overcommitment unpractical

OoH



- ✓ No nested hyp.
- ✓ Flexible light kernel module
- ✓ Overcommitment possible

Nested virt.



- ✗ Keeped nested hyp.
- ✓ Overcommitment possible

OoH illustration in this thesis

2 Virtualization Features

Intel PML

Intel SPP

4 Application' Examples

In privileged VMs (Xen's dom0)

Working set size estimation of VMs

In Unprivileged VMs (Xen's domU)

Checkpoint/Restore

Garbage Collection

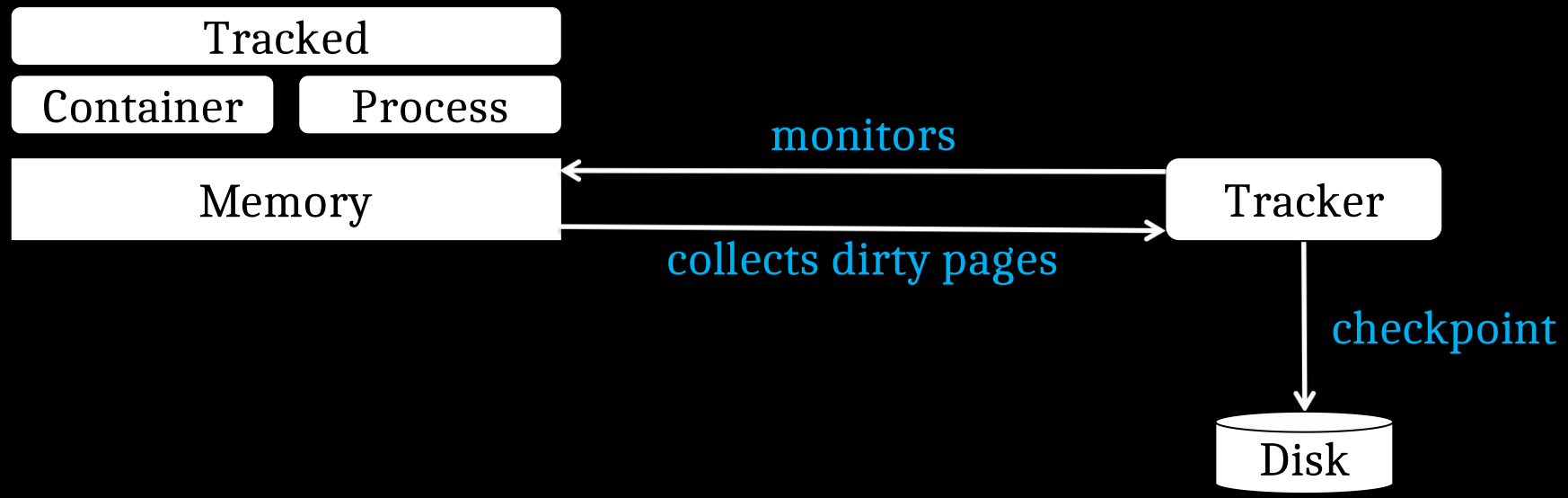
Buffer Overflow Mitigation

OoH for PML: Efficient Dirty Page Tracking In Virtualized Clouds

Dirty Page Tracking

PML is exposed to the guest for HW-assisted dirty page tracking: monitoring of dirty (or modified) pages of an application/process

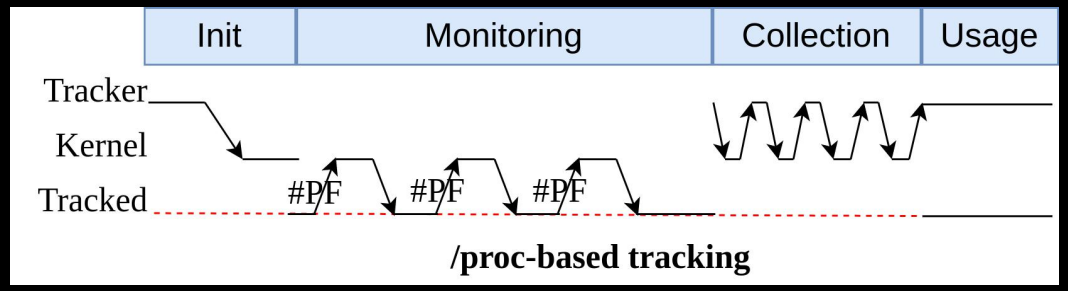
Usefulness of dirty page tracking: checkpointing (backup images for recovery after failure), garbage collection (freeing unused objects for better memory management)



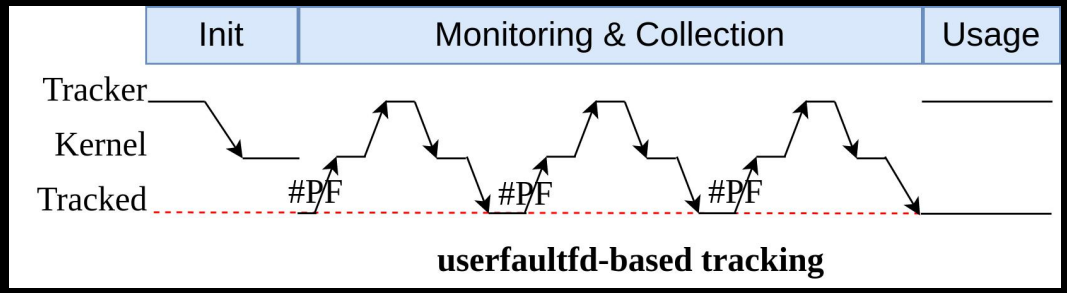
Dirty Page Tracking: Current Approach

Page write-protection
2 main interfaces in Linux

/proc



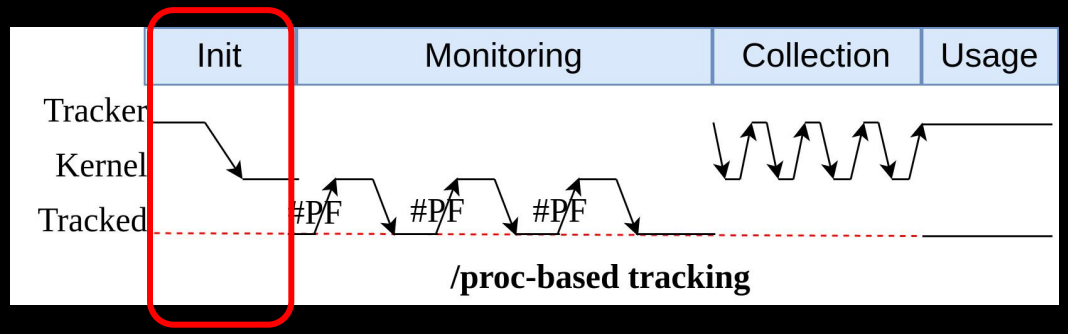
userfaultfd



Dirty Page Tracking: Current Approach

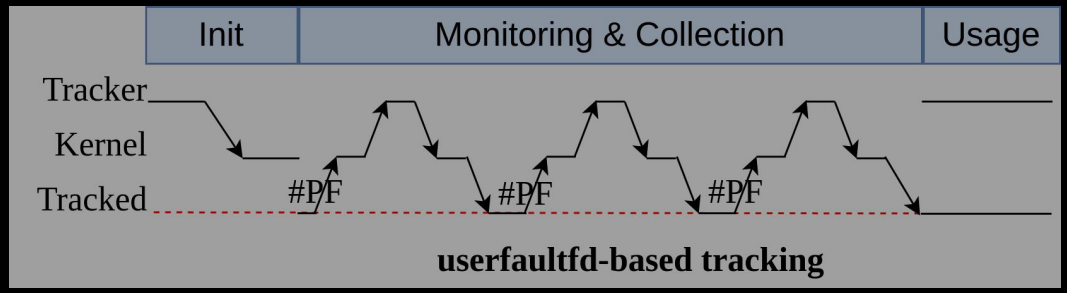
Page write-protection
2 main interfaces in Linux

/proc



syscall to invalidate all Tracked' pages

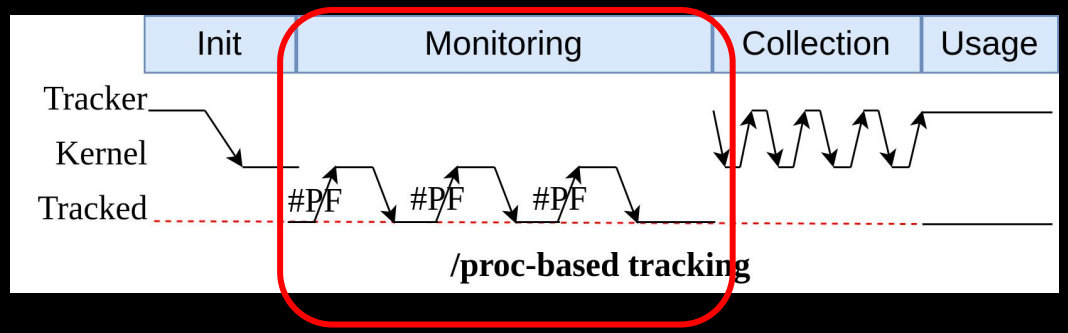
userfaultfd



Dirty Page Tracking: Current Approach

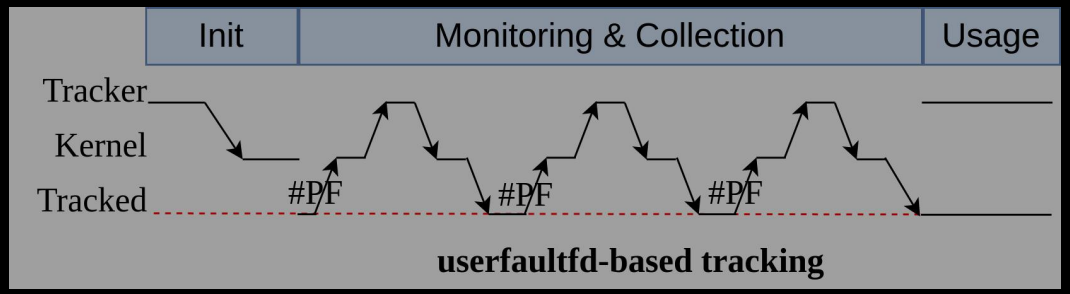
Page write-protection
2 main interfaces in Linux

/proc



page faults (#PF) generated upon access

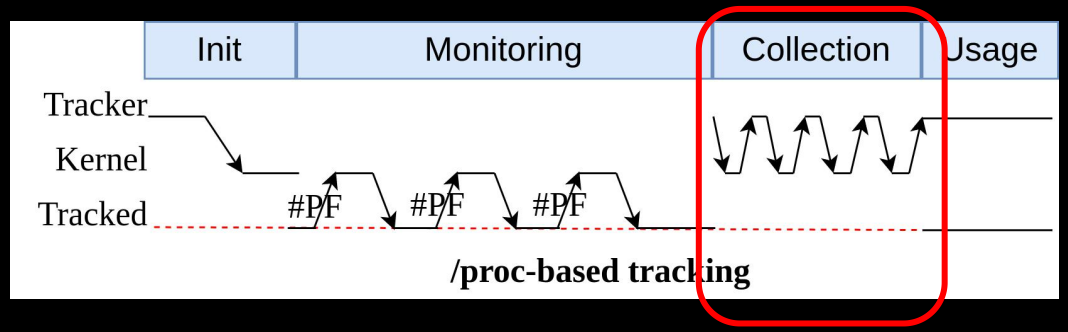
userfaultfd



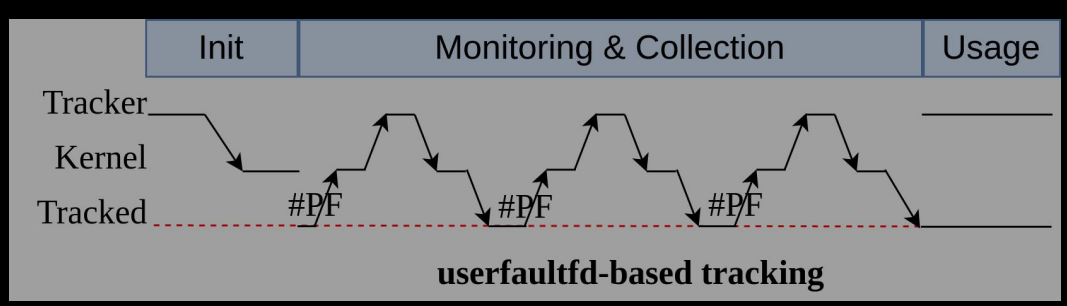
Dirty Page Tracking: Current Approach

Page write-protection
2 main interfaces in Linux

/proc



userfaultfd

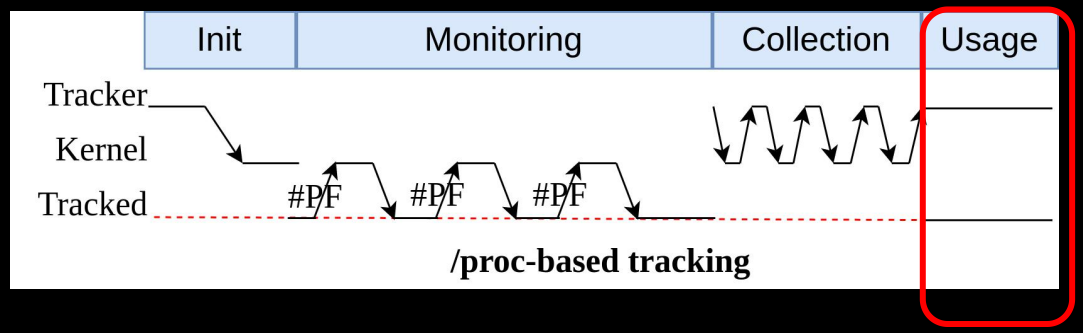


Tracker reads `/proc/PID/maps` and `/proc/PID/pagemap` (with PID of Tracked) uses the dirty bit to determine accessed pages

Dirty Page Tracking: Current Approach

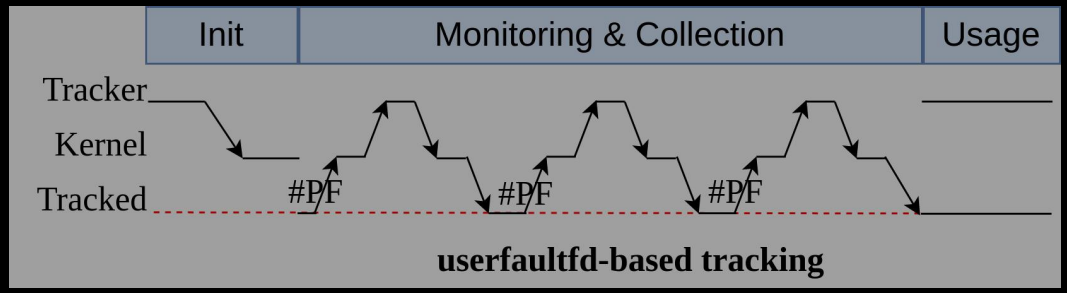
Page write-protection
2 main interfaces in Linux

/proc



e.g., for checkpointing

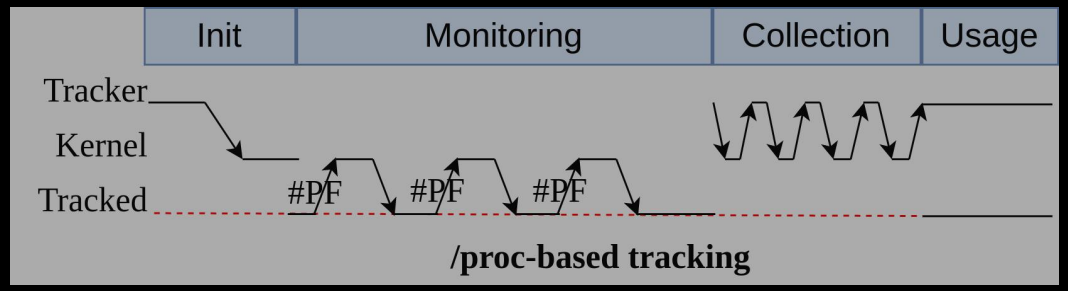
userfaultfd



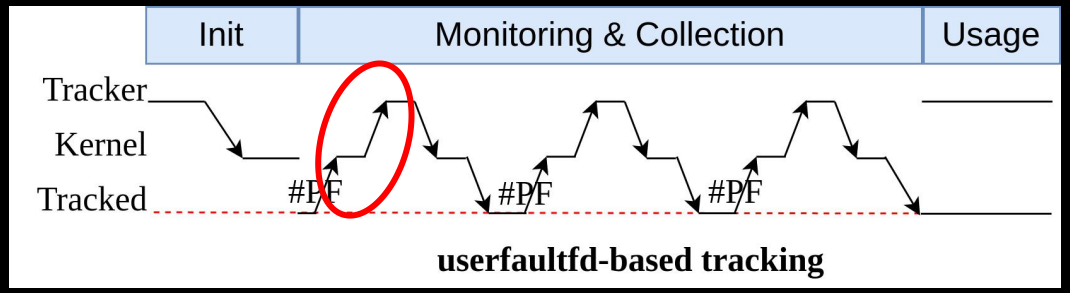
Dirty Page Tracking: Current Approach

Page write-protection
2 main interfaces in Linux

/proc



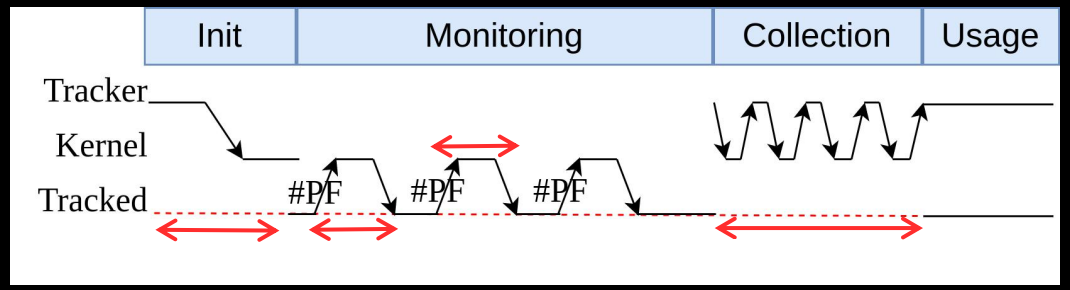
userfaultfd



the kernel transfers the #PF handling to userspace

Dirty Page Tracking: Current Approach

/proc

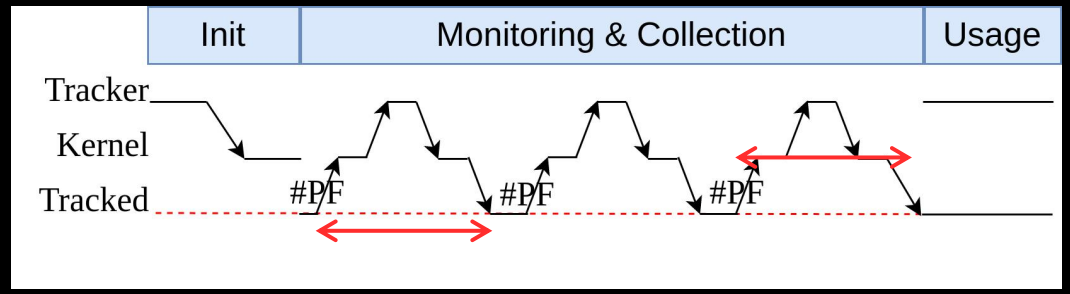


For 1GB working set:

PT walk:
in kernel (with TLB flush): ~2.23ms
in userspace (/proc/PID/pagemap): ~594.18ms

4x slowdown on Tracked
2.5x slowdown on Tracker

userfaultfd



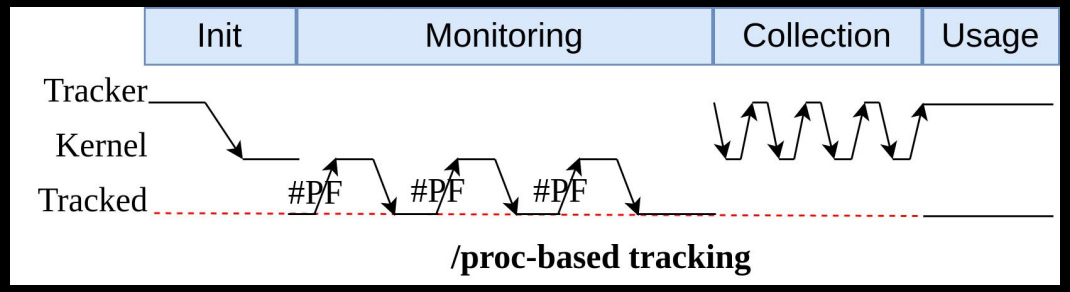
For 1GB working set:

#PF handling and context switches

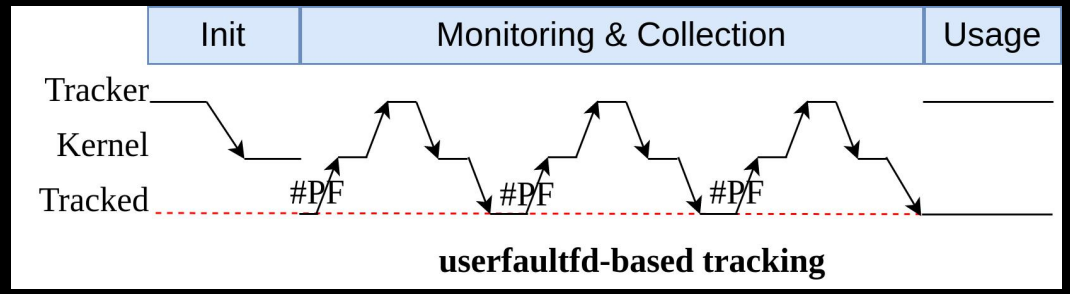
15.6x slowdown on Tracked
14.5x slowdown on Tracker

Dirty Page Tracking in Virt. Clouds: PML-based

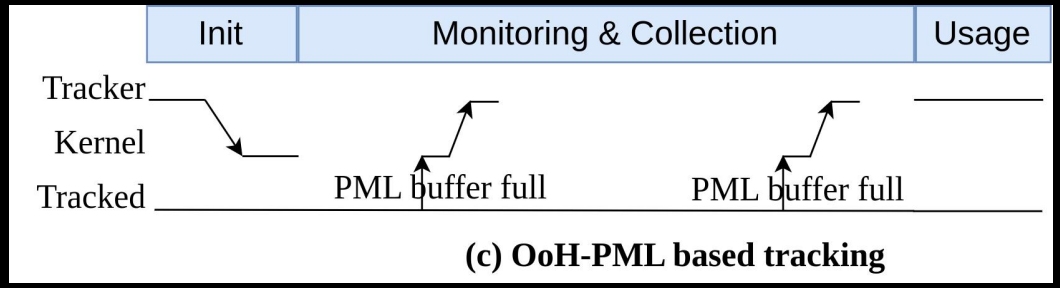
/proc



userfaultfd



PML



Challenges

(C_1) Only the hypervisor can manage PML

Challenges

(C_1) Only the hypervisor can manage PML

(C_2) PML works at coarse-grained => it concerns the entire VM

Challenges

(C_1) Only the hypervisor can manage PML

(C_2) PML works at coarse-grained => it concerns the entire VM

(C_3) PML only logs GPAs

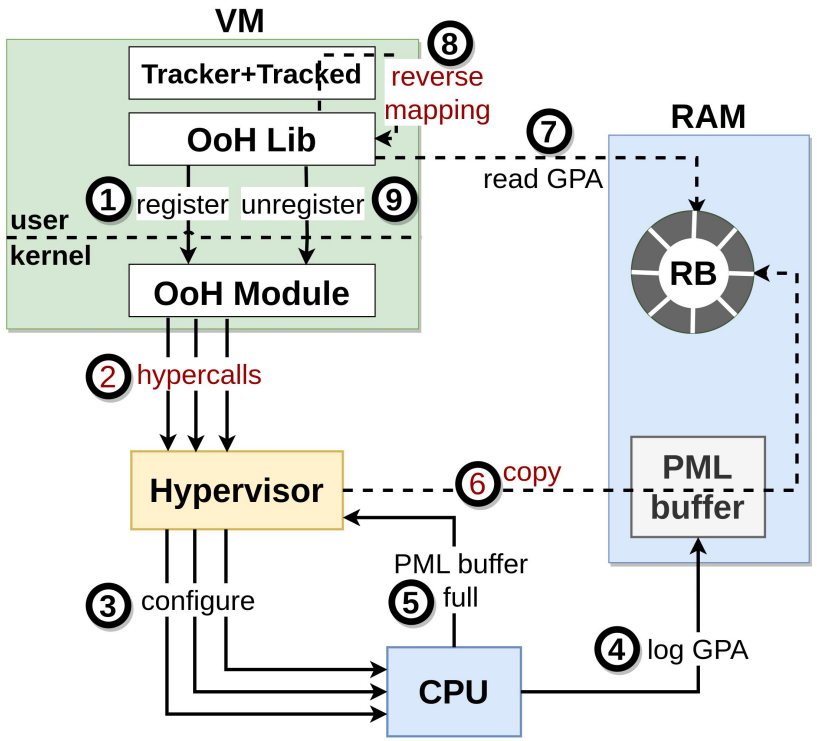
Solutions

Shadow PML (SPML)

Extended PML (EPML)

Solutions

Shadow PML (SPML)

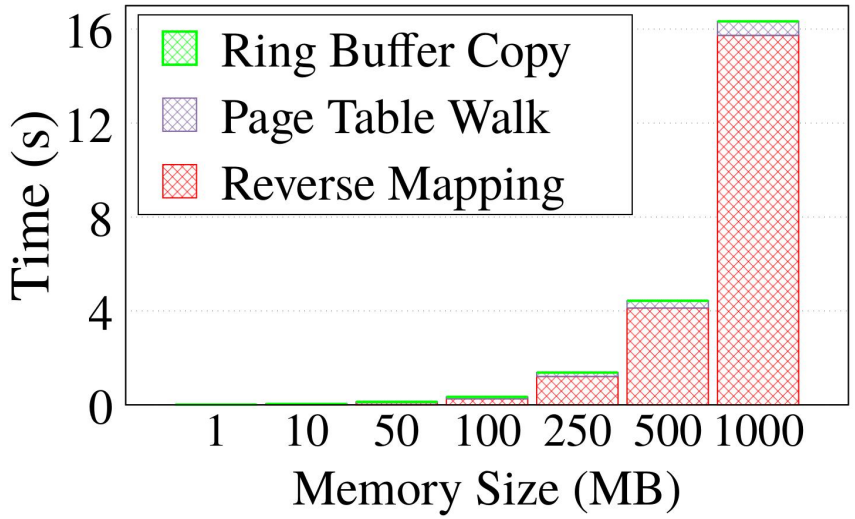


Extended PML (EPML)

Solutions

Shadow PML (SPML)

Costly reverse mapping (~15.739 s for 1GB working set)

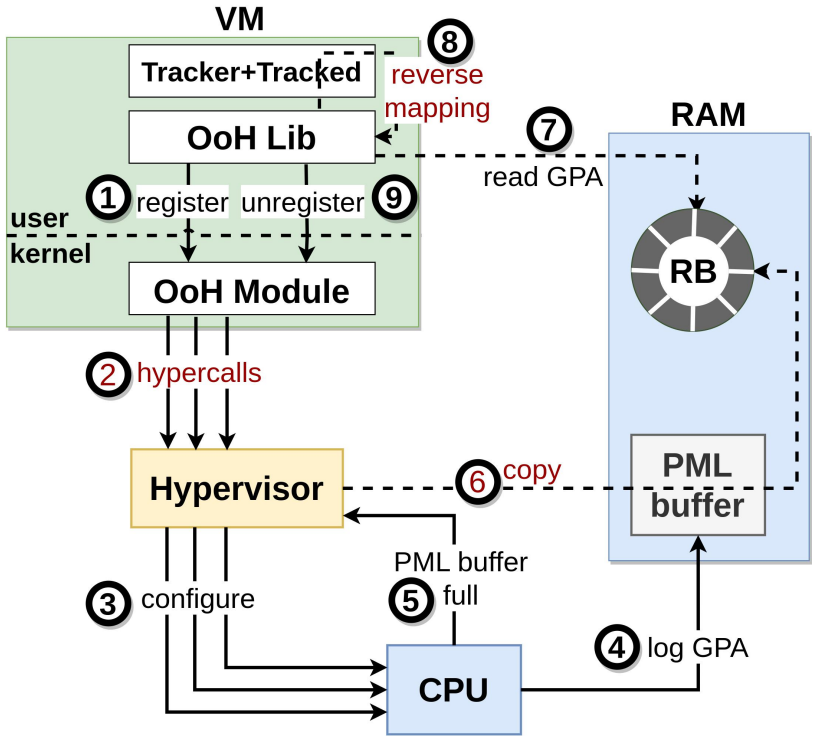


Costly hypercalls (4.49µs for an empty hypercall)

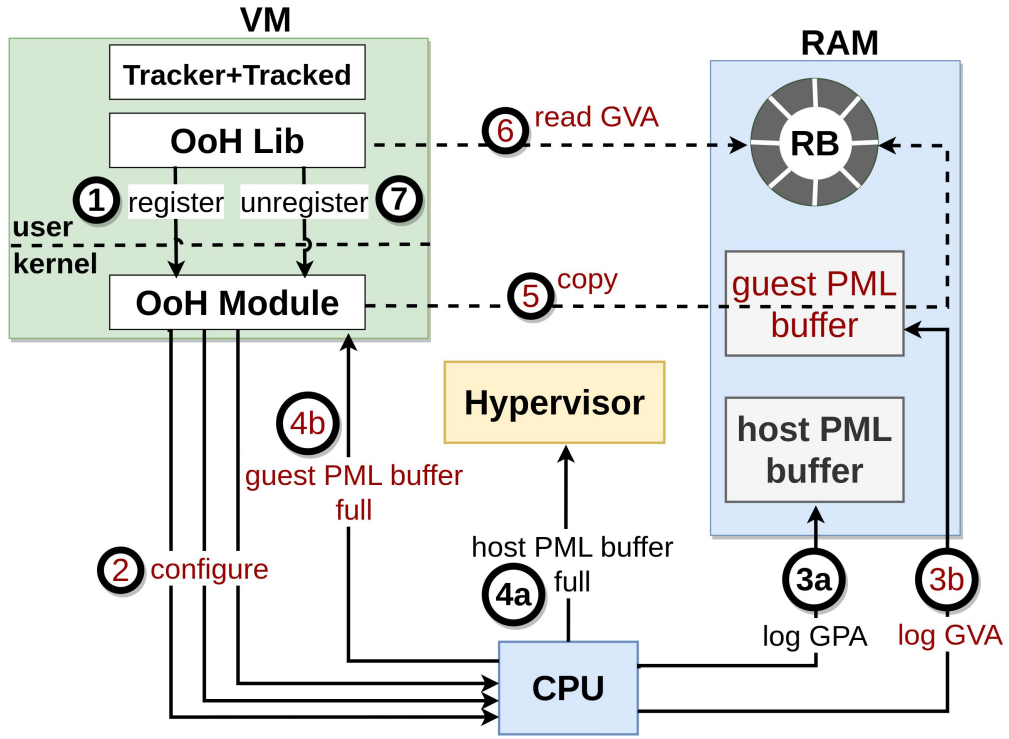
Extended PML (EPML)

Solutions

Shadow PML (SPML)



Extended PML (EPML)



Isolation & Security

Vis-à-vis the Hypervisor

- ✓ Small TCB 1 (194LOC) - at least safe as existing hypercalls
- ✓ Guest does not see nor manipulate host physical memory
- ✓ Ring buffer allocated from VM's memory

Isolation & Security

Vis-à-vis the Hypervisor

- ✓ Small TCB 1 (194LOC) - at least safe as existing hypercalls
- ✓ Guest does not see nor manipulate host physical memory
- ✓ Ring buffer allocated from VM's memory

Between VMs

- ✓ Intrinsic VM' isolation not altered
- ✓ Ring buffer allocated per VM's address space => no possible inference
- ✓ Per process ring buffer and restriction to tracker process only

Implementation

EPML's hardware changes under BOCHS
- ✓ EPML: 44 LOCs in 6 files

Xen as the hypervisor
- ✓ SPML: 182 LOCs in 13 files
- ✓ EPML: 120 LOCs in 9files

Linux as the guest OS
- ✓ SPML: 6 LOCs in 2 files
- ✓ EPML: 14 LOCs in 9files

OoH Module:
- ✓ SPML: +520 LOCs
- ✓ EPML: +520 LOCs

Integration of the OoH Lib with:

- ✓ CRIU: Checkpoint/Restore in User space (used in OpenVZ, Docker, etc.)
  - Based on /proc technique
  - SPML: 251 LOCs in 4 files
  - EPML: 140 LOCs in 4 files

- ✓ Boehm GC: popular C/C++ garbage collector (used in Mozilla, GNU Java Compiler, etc.)
  - Based on /proc technique
  - SPML: 254 LOCs in 4 files
  - EPML: 144 LOCs in 4 files

Evaluation Methodology

Build a formula to evaluate EPML
Show its accuracy on other techniques that are measurable

Impact on Tracker: Execution time of Tracker when implementing technique x

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

x : /proc, SPML, EPML - C x : enable_PML, ring buffer copy, etc.

Impact on Tracked: Time of Tracked when monitored by a Tracker using technique x

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

$I(C_x, C_{tked})$: page faults, vmexits, etc., etc.

Evaluation Methodology: Validtion

SPML

Metric	Time (ms)
$E(C_{tker})$ measured	5503.79
$E(C_{tked\_tker})$ measured	135255.35
$E(C_{p})$	251.35
$E(C_{copy\_rb})$	0.49
$E(C_{disable\ pml})$	2.06
$E(C_{rev.\ mapping})$	5419
$E(C_{tker})$ estimated	5672.9
$E(C_{vmexits})$	18000
N	39
$E(C_{vmread,vmwrite})$	$1.73x10^{-3}$
$E(C_{tked\_tker})$ estimated	136919.85

/proc

Metric	Time (ms)
$E(C_{tker})$ measured	1097.99
$E(C_{tked\_tker})$ measured	115283.35
$E(C_{p})$	251.35
$E(C_{clear\_refs})$	1.409
$E(C_{PT\ walk})$	0.89
$E(C_{tker})$ estimated	1116.09
$E(C_{PFHkernel})$	0.27
$E(C_{tked\_tker})$ estimated	114418.58

Evaluation Methodology: Validtion

SPML

Metric	Time (ms)
E(C_{tker}) measured	5503.79
E(C_{tked_tker}) measured	135255.35
E(C_p)	251.35
E($C_{..py\_rb}$)	0.49
E($C_{..disable\ pml}$)	2.06
E($C_{rev.\ mapping}$)	5419
E(C_{tker}) estimated	5672.9
E($C_{vmexits}$)	18000
N	39
E($C_{vmread,vmwrite}$)	$1.73x10^{-3}$
E(C_{tked_tker}) estimated	136919.85

96.34% accuracy

/proc

Metric	Time (ms)
E(C_{tker}) measured	1097.99
E(C_{tked_tker}) measured	115283.35
E(C_p)	251.35
E(C_{clear_refs})	1.409
E($C_{PT\ walk}$)	0.89
E(C_{tker}) estimated	1116.09
E($C_{PFHkernel}$)	0.27
E(C_{tked_tker}) estimated	114418.58

99% accuracy

Evaluation Benchmarks

Benchmarks

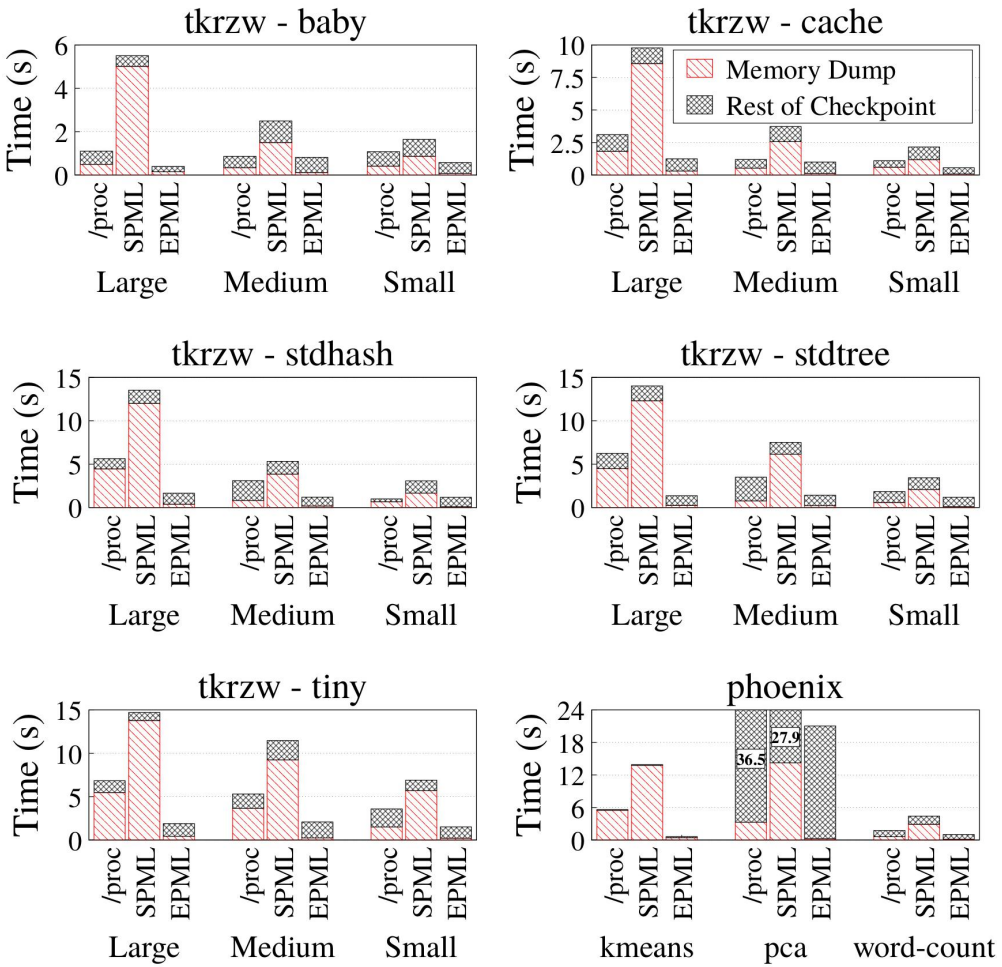
Macro-benchmarks: tkrzw applications (key value store) and Phoenix applications MapReduce)
Three working set sizes (Small, Medium, and Large)

Metrics

Execution times of tracker and tracked
Evaluations aim answering the following questions:

1. What is the potential overhead or improvement of SPML and EPML compared to existing solutions (/proc and ufd)?
2. What is the scalability of SPML and EPML?

Evaluation Results with CRIU: for Tracker



Mem dump (MD) phase: during which CRIU collects dirty pages

- /proc => walk /proc/$pid/(maps, pagemap)
- SPML => copy ring buffer + reverse mapping
- EPML => copy via ring buffer

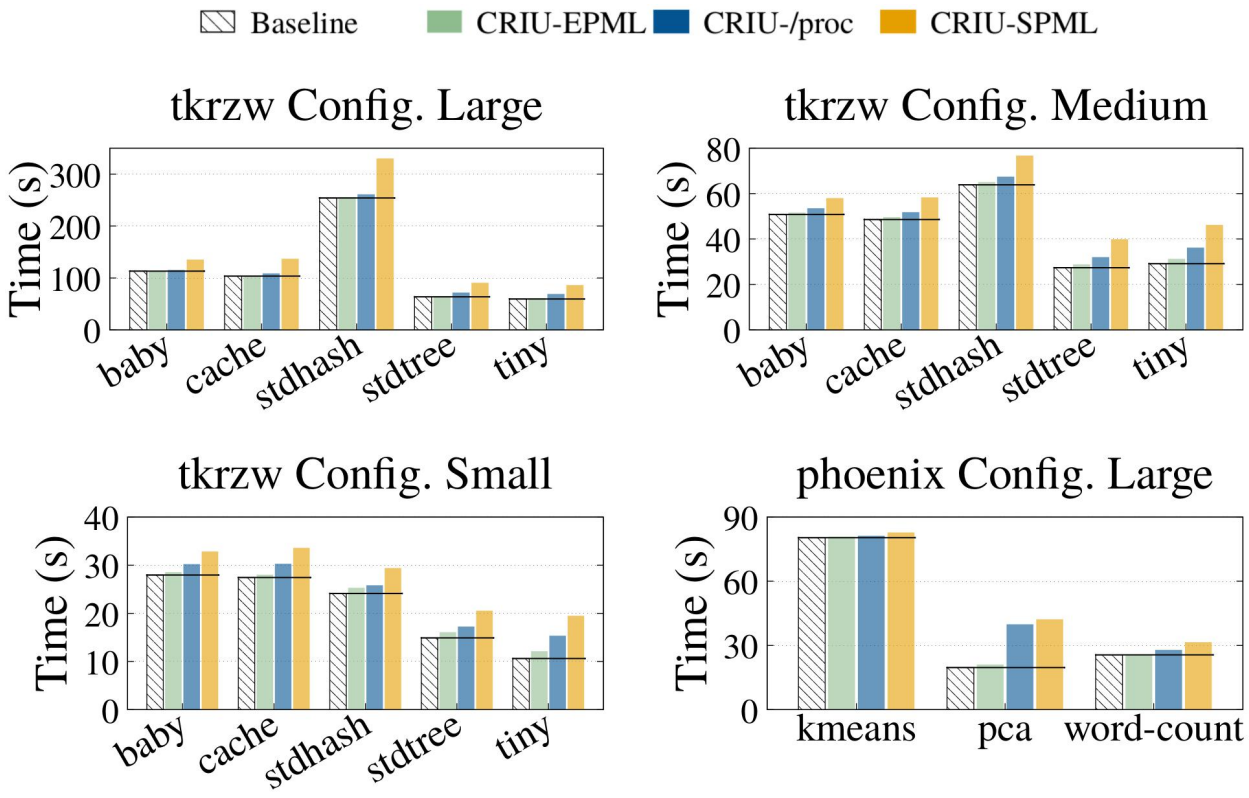
SPML:

- reverse mapping is +66% of MD time
- **5X** slowdown vs. /proc for tkrzw-baby and phoenix-kmeans (config. Large)

EPML:

- **13X** speedup vs. SPML for tkrzw-baby (config. Large)
- **4X** speedup vs. /proc for tkrzw-baby (config. Large)

Evaluation Results with CRIU: for Tracked



CRIU pauses applications during checkpointing resulting in:

SPML:
- Up to 102% overhead on Phoenix-pca with CRIU
- 34.6% overhead on average

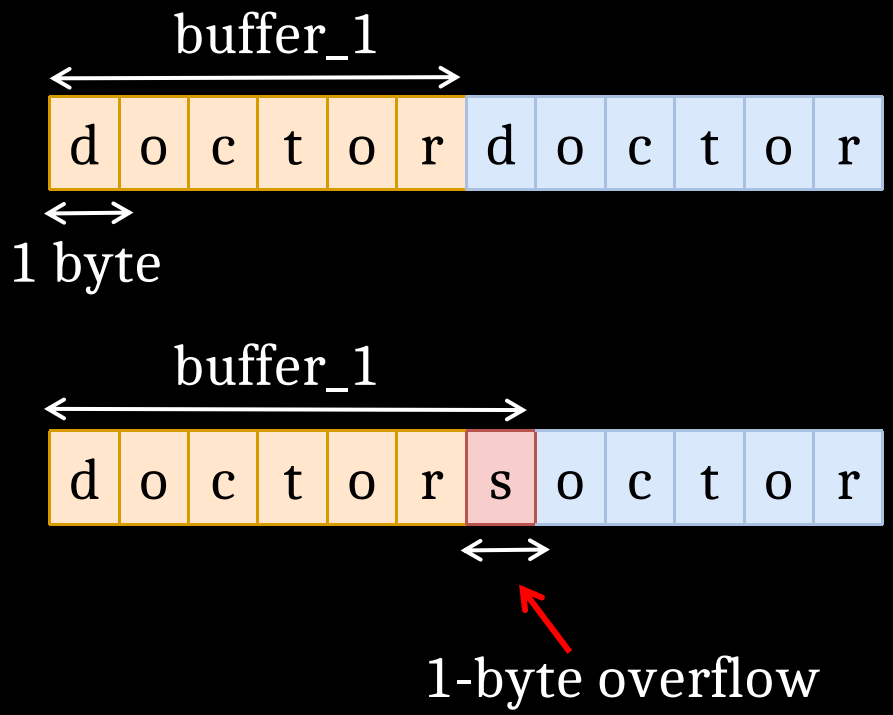
/proc:
- Up to 114% overhead on Phoenix-pca with CRIU
- 15.6% overhead on average

EPML:
- Only 7% overhead on Phoenix-pca with CRIU
- 3.5% overhead on average
- avg. +9x and +4x improvement on SPML & /proc resp.

OoH for SPP: Efficient Buffer Overflow Detection In Virtualized Clouds

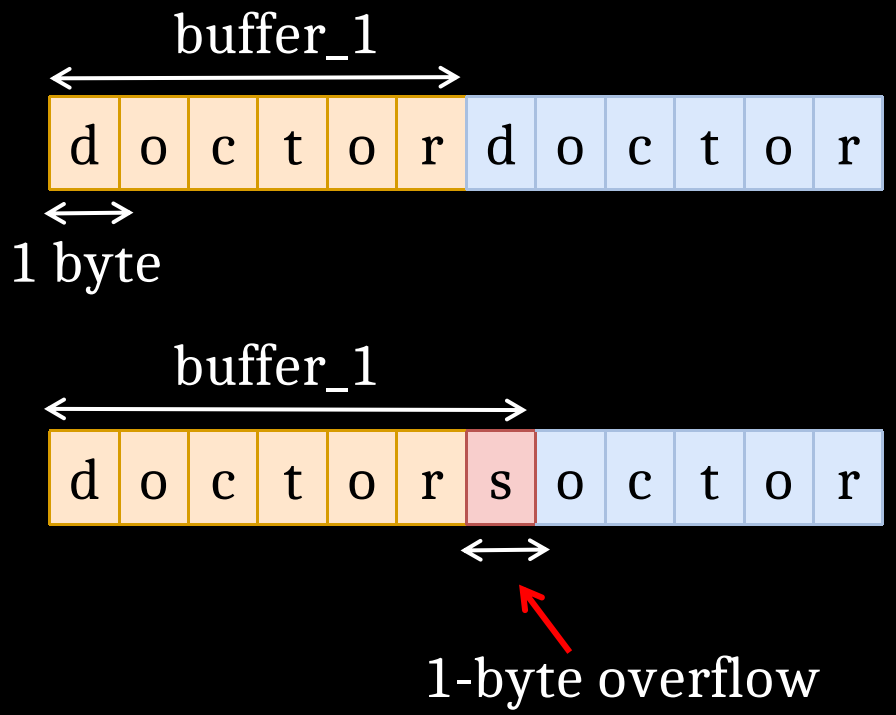
Buffer Overflow: Introduction

Definition



Buffer Overflow: Introduction

Definition



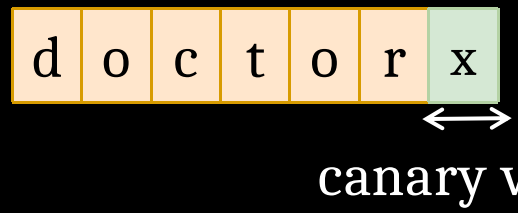
Importance

- ✓ Can lead, for example, to DB corruption (e.g., if overflow modifies a password), carrying out an attack (e.g., malware code injection), etc.
- ✓ 70% Google Chrome's bugs [50, Google security blog]
- ✓ Most prevalent vulnerability in 2022 -- sources [51, CWE (Common Weaknesses and Exposure)]

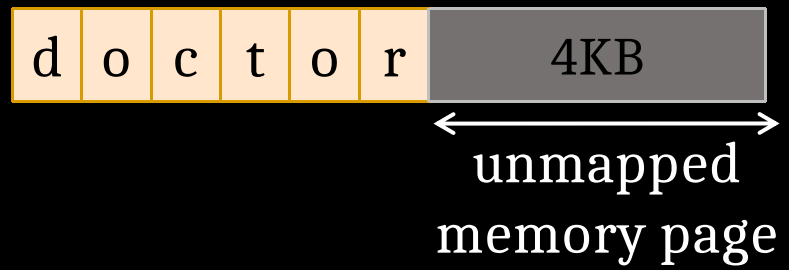
Buffer Overflow: State-of-the-art Mitigation Mechanisms

Detection Mechanisms

Canary



Guard Page



Hardware Capabilities

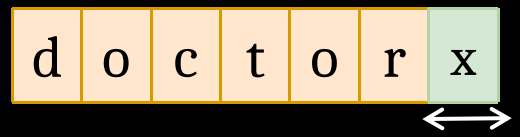
CHERI: Capabilities Hardware Enhance RISC-V Instructions [159, ARCH 2014]

- Overflow detection by the HW
- Needs to rebuild systems [156, Tech. Rep. 2020]
- Incurs significant overheads (up to 250%) [156, Tech. Rep. 2020]

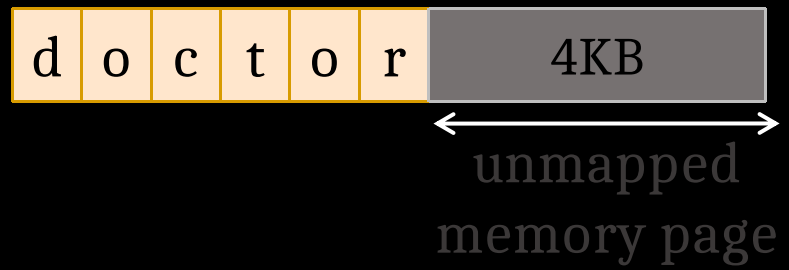
Buffer Overflow: State-of-the-art Mitigation Mechanisms

Detection Mechanisms

Canary



Guard Page



Hardware Capabilities

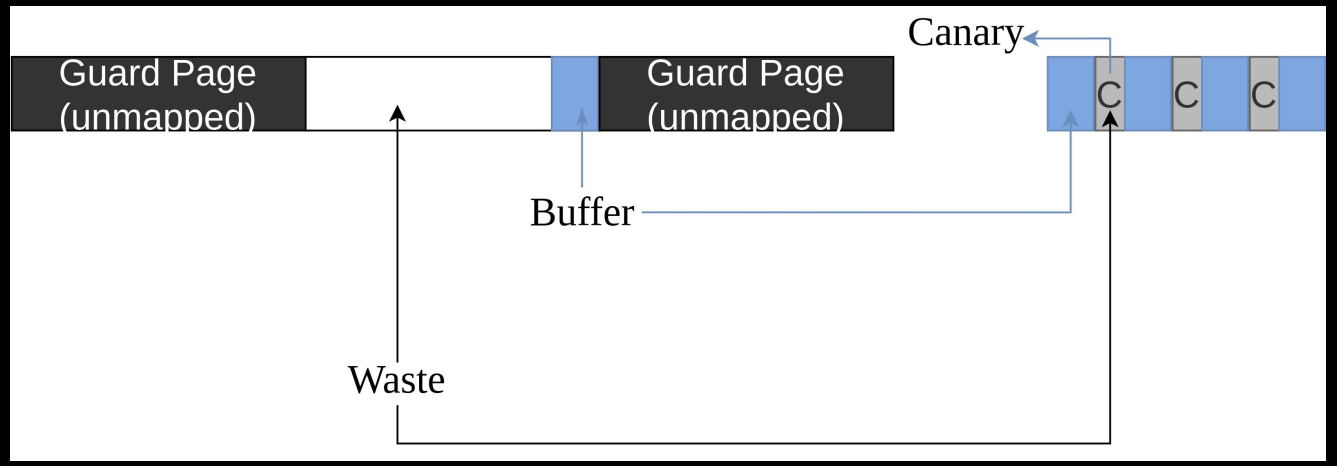
CHERI: Capabilities Hardware Enhance RISC-V Instructions [159, ARCH 2014]

- Overflow detection by the HW
- Needs to rebuild systems [156, Tech. Rep. 2020]
- Incurs significant overheads (up to 250%) [156, Tech. Rep. 2020]

Synchronuous Detection vs Memory Overhead

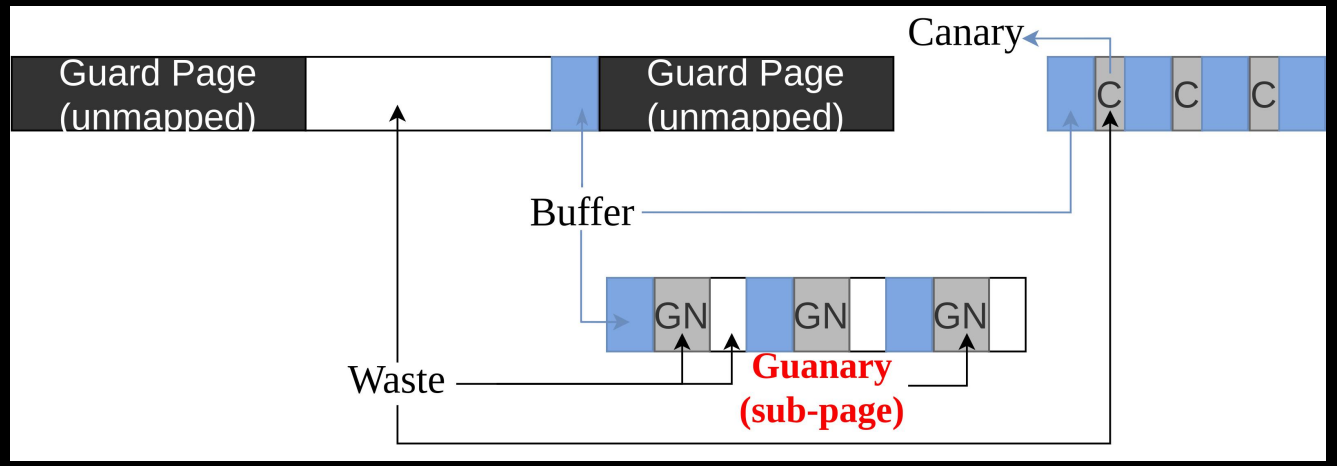
Canary
modest memory overhead
asynchronous detection

Guard page
significant memory overhead
synchronous detection

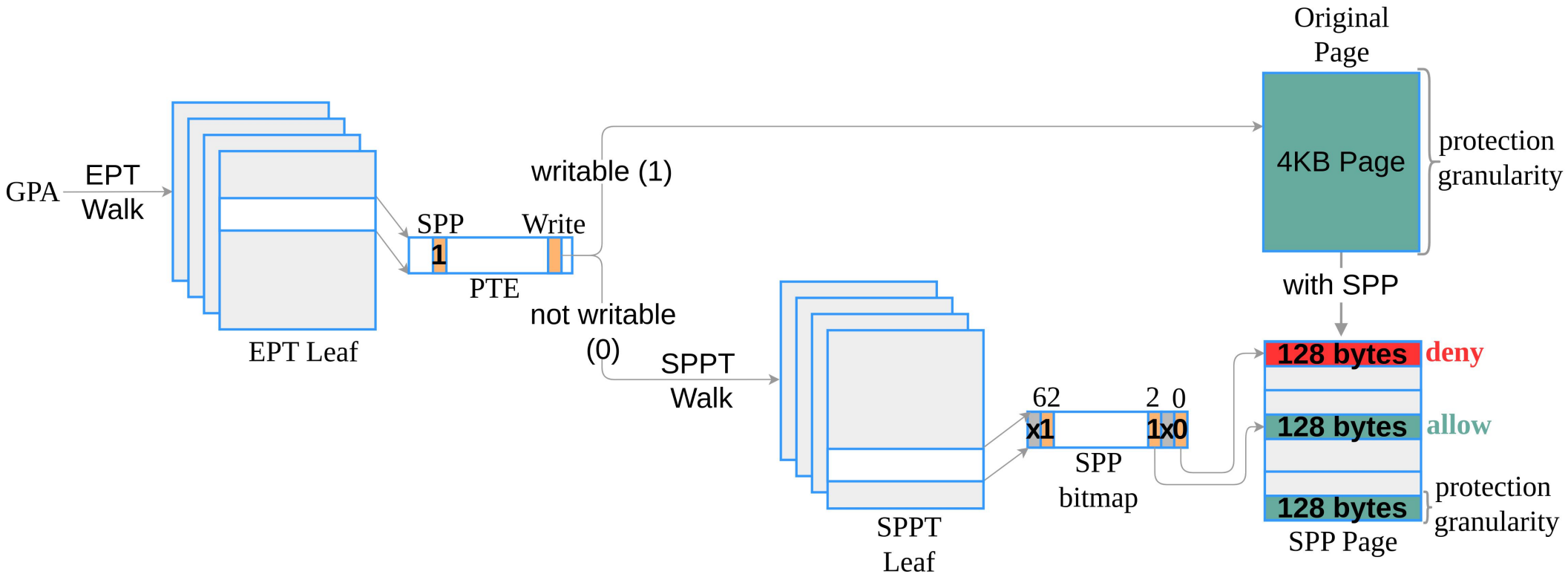


Synchronuous Detection vs Memory Overhead

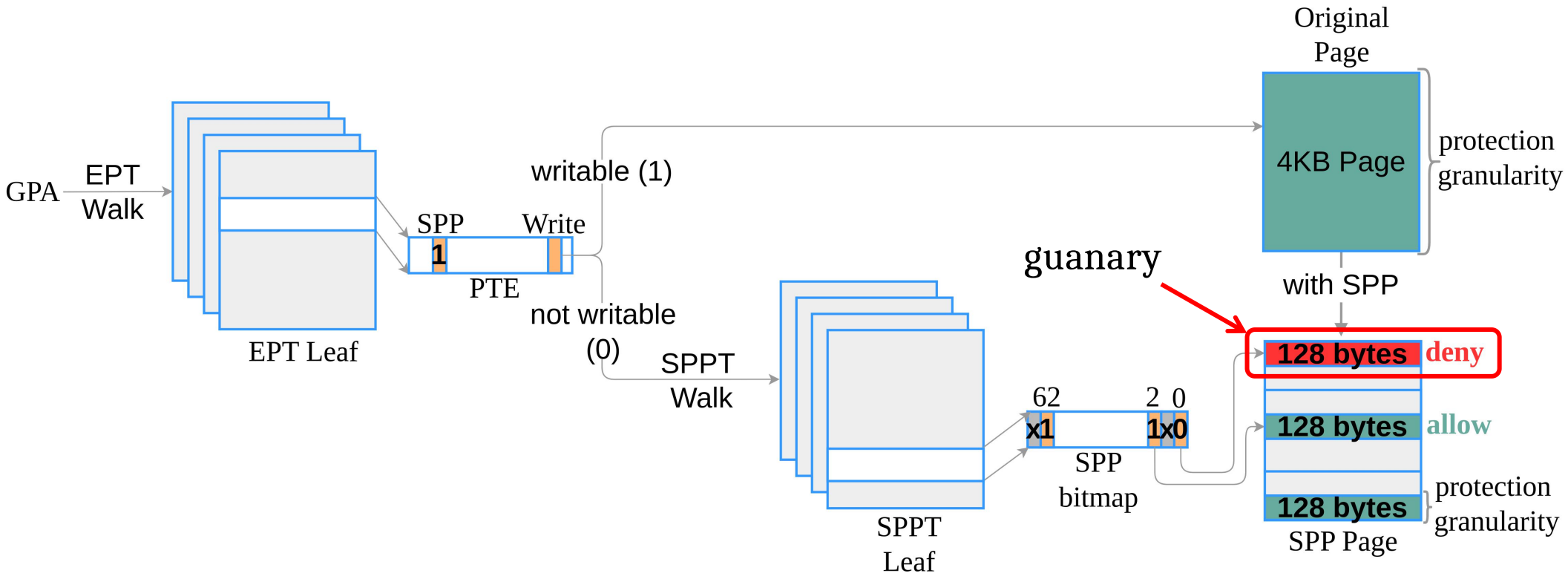
Canary	GuaNary	Guard page
modest memory overhead	modest memory overhead	significant memory overhead
asynchronous detection	synchronous overflow detection	synchronous detection



GuaNary: Intel Sub-Page write Permissions (SPP)



GuaNary: Intel Sub-Page write Permissions (SPP)



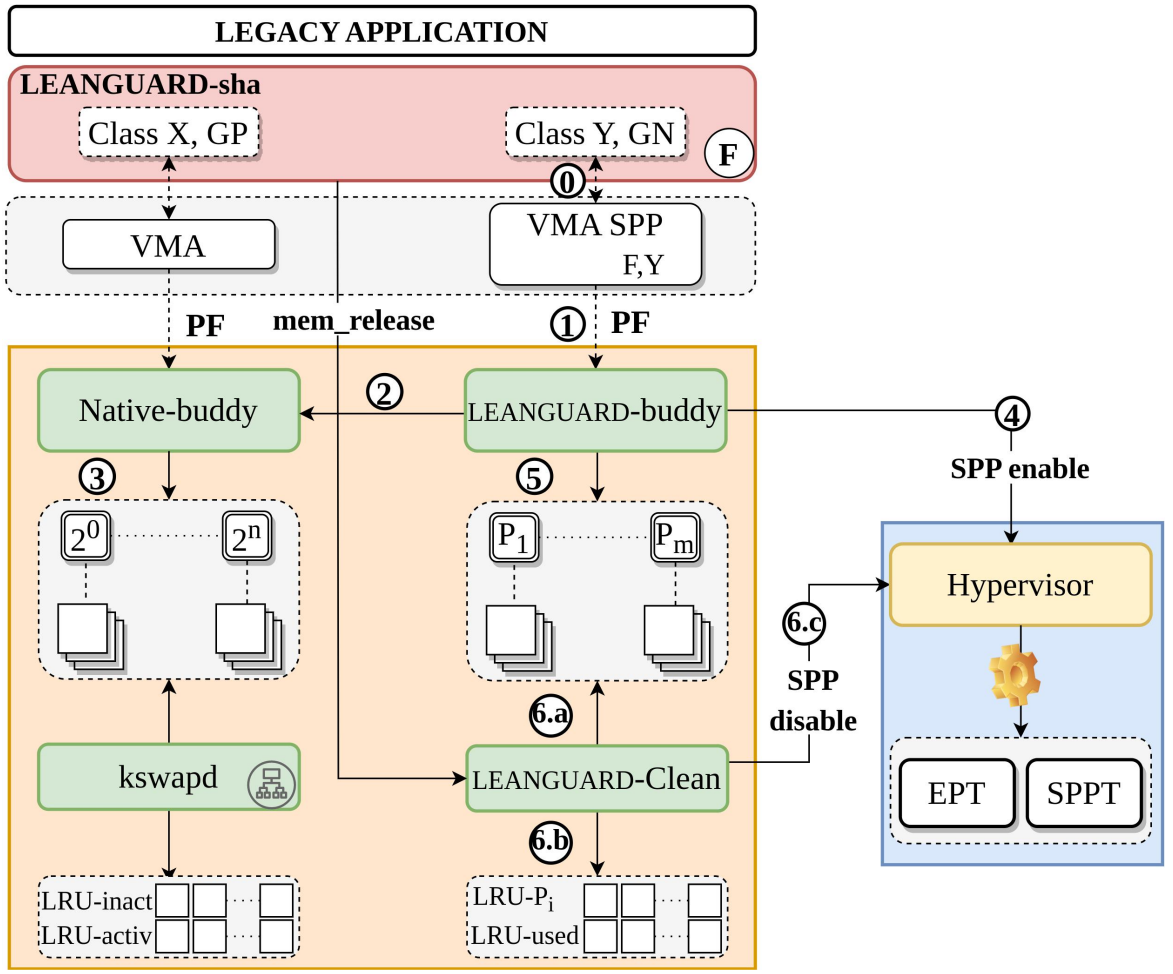
GuaNary: Challenges

Can be summarized in 2 main:

Conceptual: Costly hypercalls and EPT+SPPT walk

Technical: Page heterogeneity

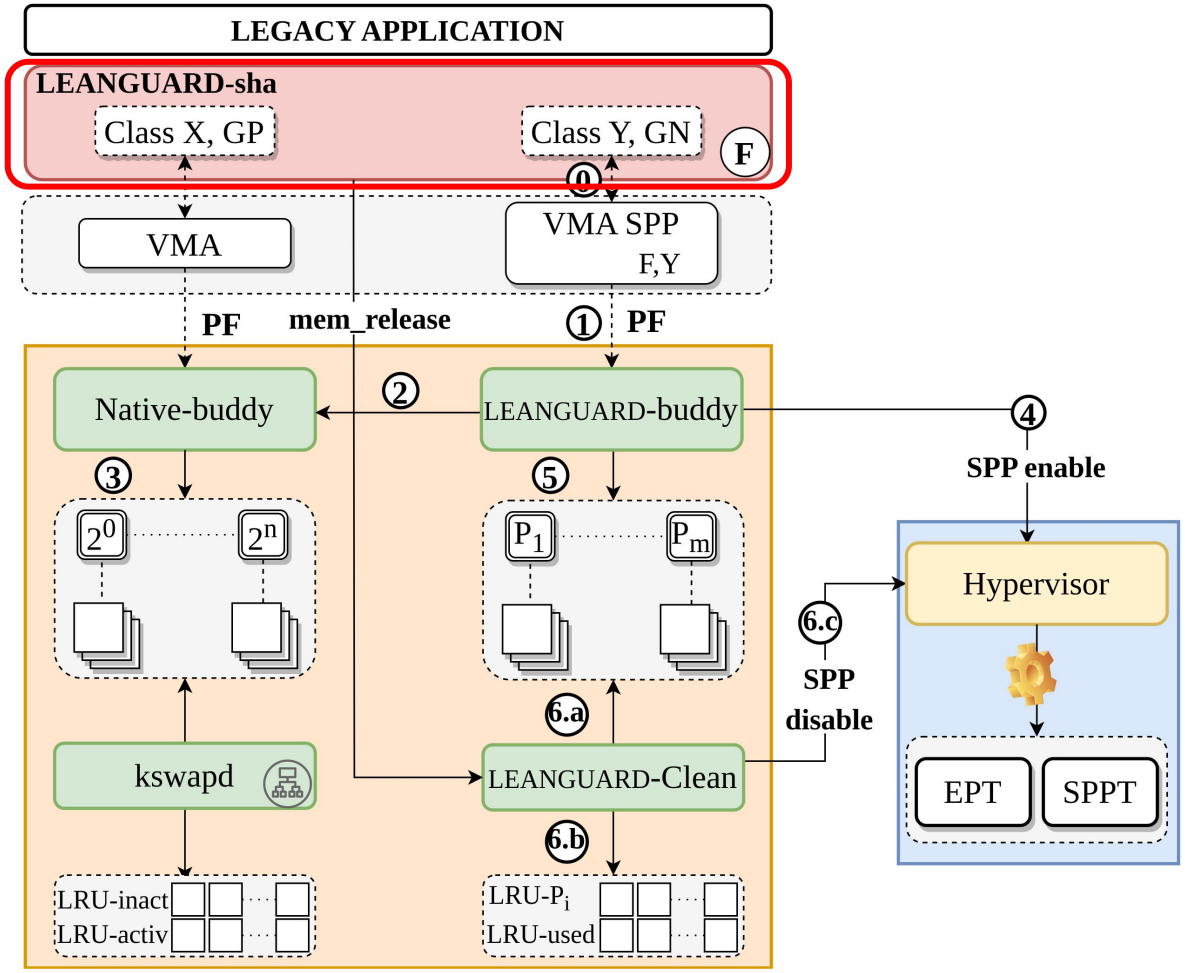
LeanGuard: Complete SW Stack to use Guanary



LeanGuard: Complete SW Stack to use Guanary

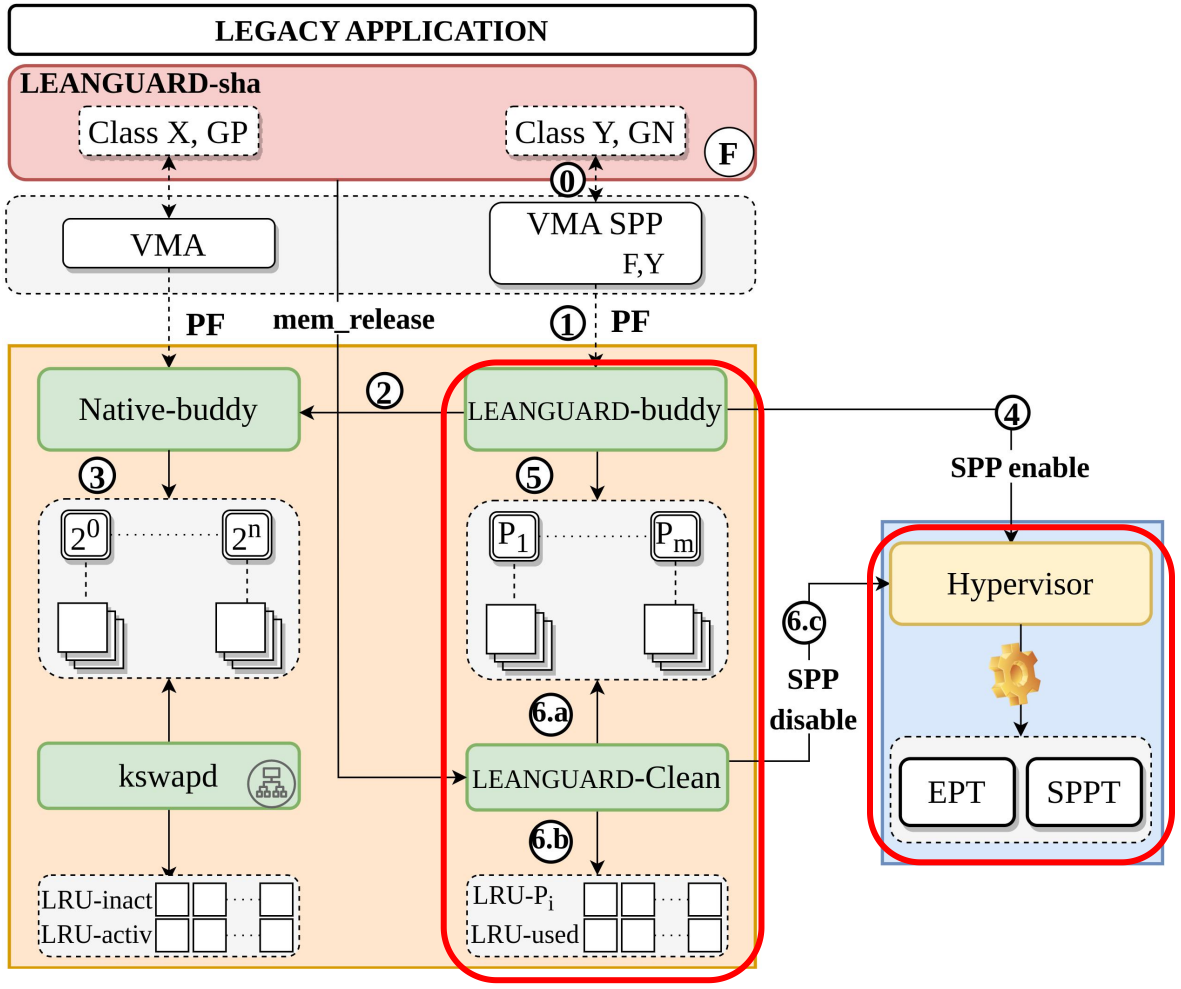
LeanGuard-sha configures 2 param:

- G: type of guardian (GP or GN)
  - GP if sizeof(class) % 4KB = 0
  - Guanary if not
- F: protection frequency (configured by the user)



LeanGuard: Complete SW Stack to use Guanary

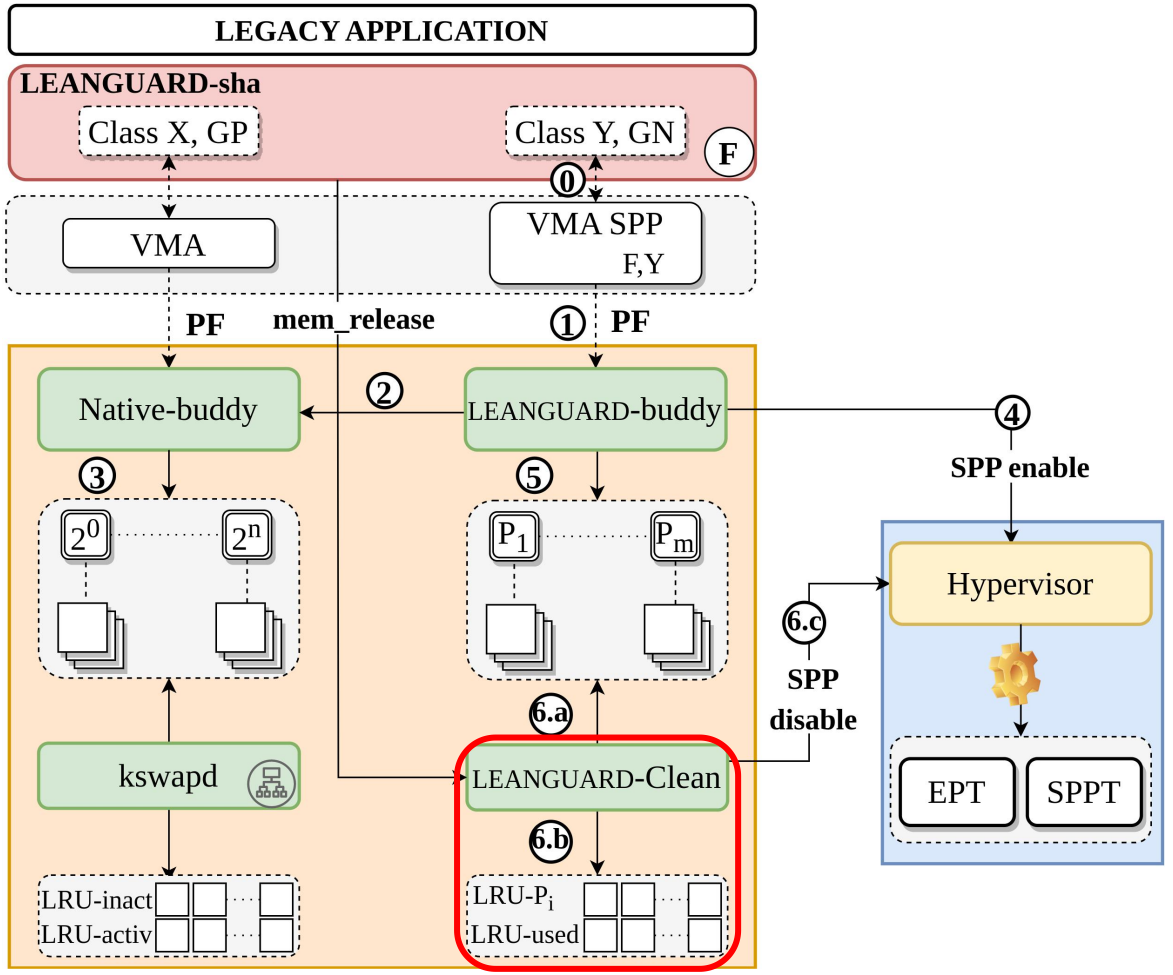
- New buddy allocator responsible for SPP page allocation
- Batching hypercalls for optimal performance
- Hypervisor optimization: SPP config for a group of contiguous pages



LeanGuard: Complete SW Stack to use Guanary

Memory reclaim: LeanGuard-Cleaner

- Handles SPP page release on process termination
  - Reinserts SPP pages into corresponding pool
  - Returns the excess of pages to Native-buddy for all pools: batch hypercall for SPP deactivation



LeanGuard: Implementation

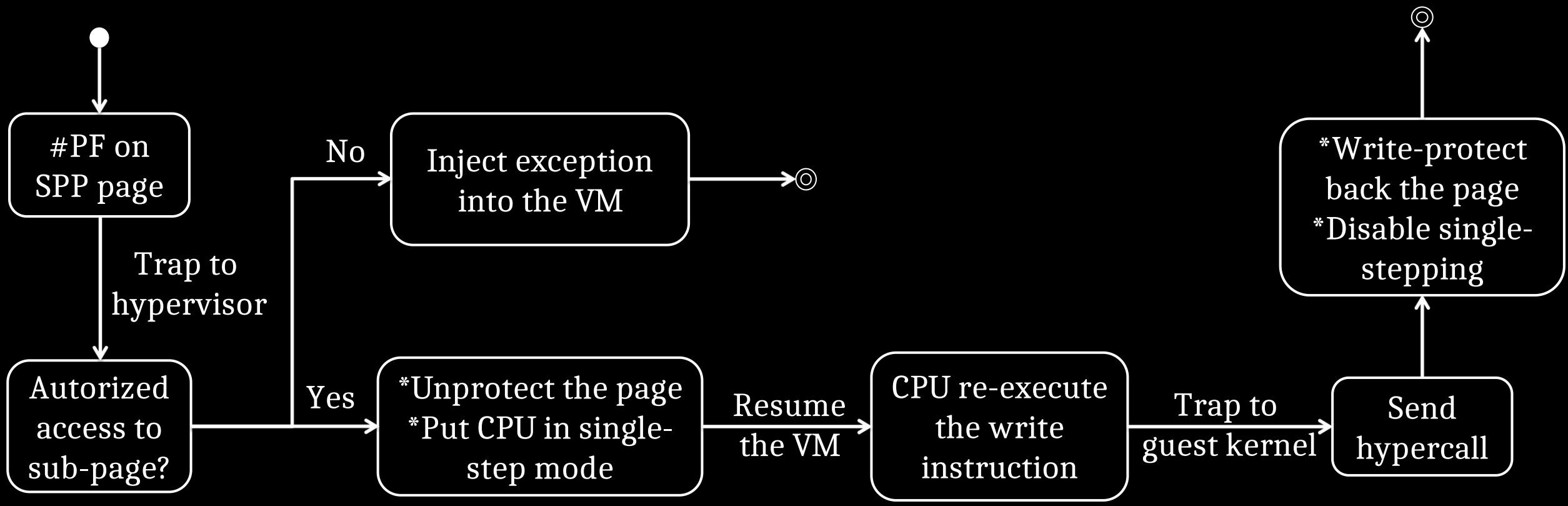
Xen hypervisor version 4.10: 600LOCs addition

Linux OS version 5.11.14: 750LOCs addition

SlimGuard secure allocator: 100LOCs addition

- 2019 State-of-the-art BIBOP (Big Bag Of Pages) secure allocator
- Proven more memory-efficient than other state-of-the-art BIBOP allocators (Guarder, FreeGuard, etc.)

LeanGuard Evaluation: Memory Overhead Methodology



LeanGuard Evaluation

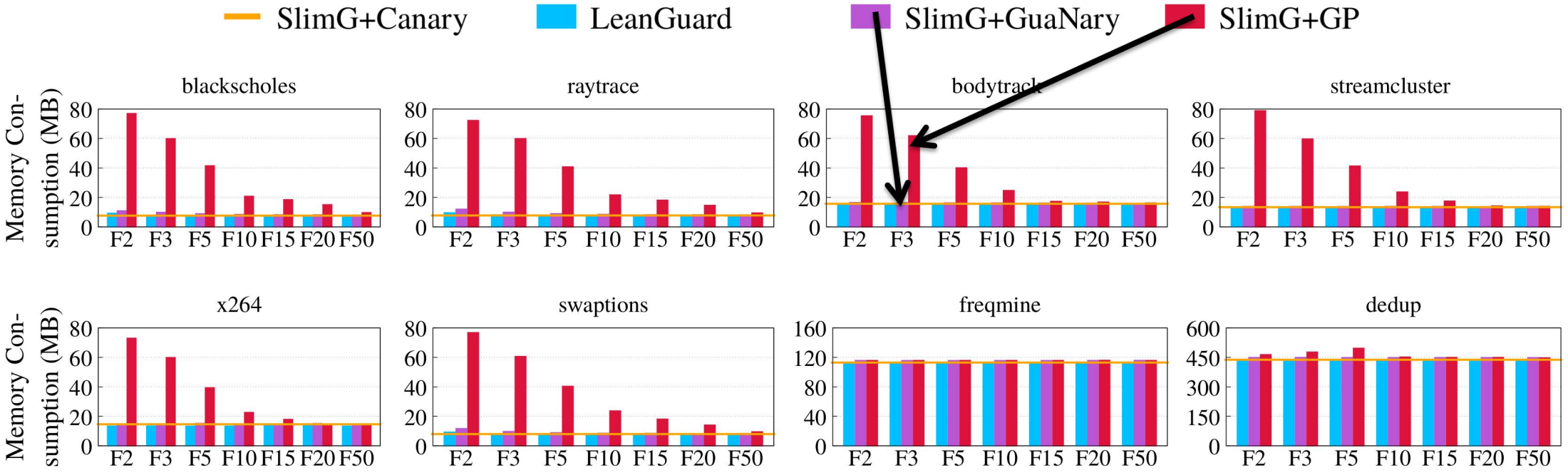
Benchmarks

PARSEC applications: blackscholes, bodytrack, dedup, fluidanimate, freqmine, raytrace, streamcluster, swaptions, and x264

Our experiments hinge on:
Memory overhead and Security trade-off
Performance overhead and scalability

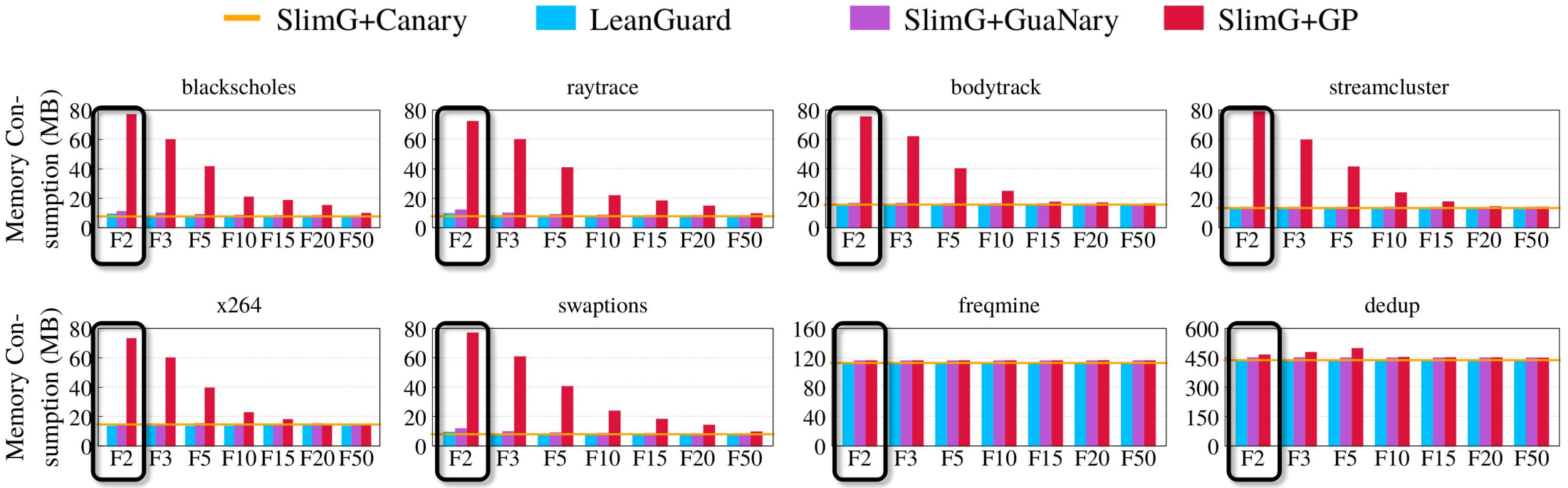
LeanGuard's goal is minimizing memory overhead while allowing synchronous buffer overflow detection

LeanGuard Evaluation: Memory Overhead Results



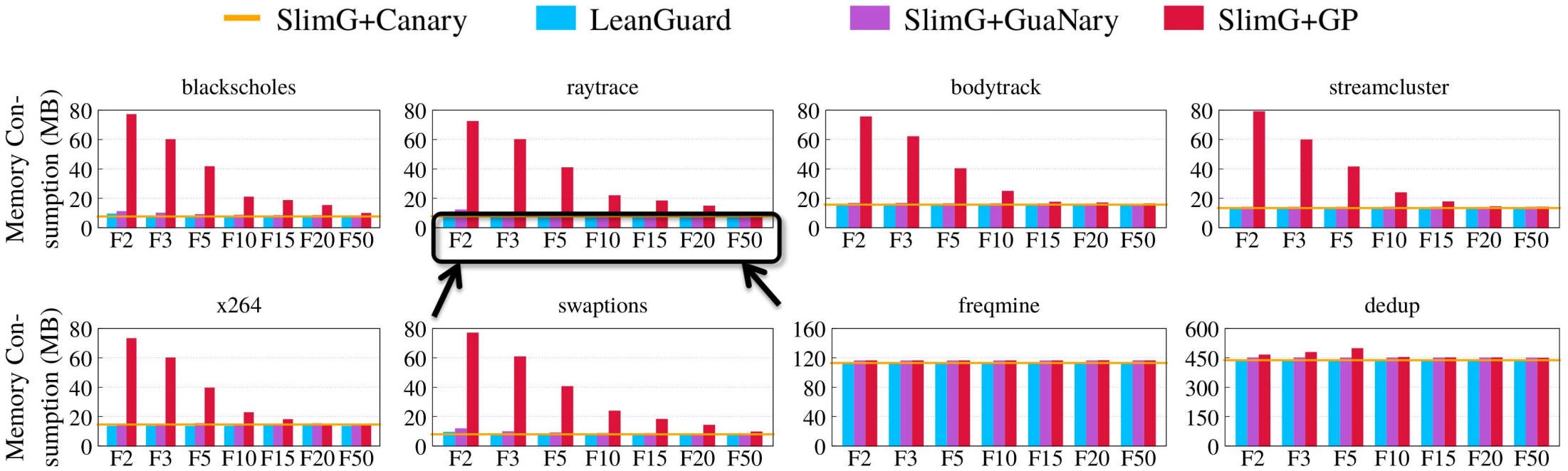
GuaNary effectively leads to memory consumption reduction
Slight improvement that LeanGuard brings to SlimG+GuaNary (26% swaptions, 25% on raytrace, and 18% blacksholes, when F=2) is explained by the fact that, unlike SlimG+GuaNary that exclusively uses GuaNary, LeanGuard can use 4KB guard pages when necessary

LeanGuard Evaluation: Memory Overhead Results



SlimG+GP incurs a significant memory overhead compared to LeanGuard. For example, to protect 50% of the allocated buffers (F=2), LeanGuard, on average, uses 60% less memory than SlimG+GP

LeanGuard Evaluation: Memory Overhead Results



Using the same amount of memory as SlimGuard, LeanGuard allows protecting 25× more buffers than SlimG+GP

LeanGuard Evaluation: Performance Overhead Methodology

$$T_{est} = T_{noSPP} + \underbrace{TLB_{SPP} \times 4 \times T_{mem\_latency}}$$

Capture the SW overhead

Capture the HW-part overhead

LeanGuard Evaluation: Performance Overhead Methodology

$$T_{est} = T_{noSPP} + TLB_{SPP} \times 4 \times T_{mem\_latency}$$

↓

#TLB_misses due to SPP pages

LeanGuard Evaluation: Performance Overhead Methodology

$$T_{est} = T_{noSPP} + TLB_{SPP} \times 4 \times T_{mem\_latency}$$

SPPT walk requires 4 memory accesses

Time of memory access (~111ns)

LeanGuard Evaluation: Performance Overhead Methodology

$$T_{est} = T_{noSPP} + TLB_{SPP} \times 4 \times T_{mem\_latency}$$

$$= k \times TLB_{tot}$$

total number of #PFs
obtained using the Linux' perf tool

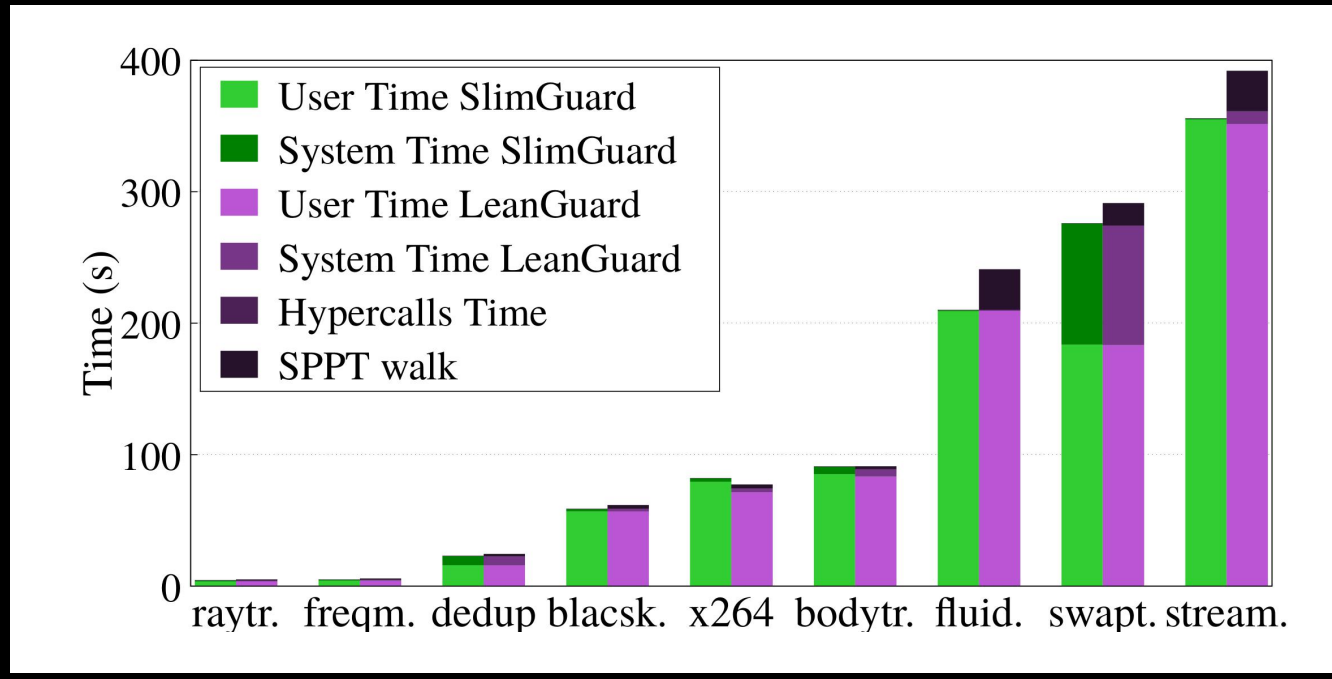
LeanGuard Evaluation: Performance Overhead Methodology

$$T_{est} = T_{noSPP} + TLB_{SPP} \times 4 \times T_{mem\_latency}$$

$$= k \times TLB_{tot}$$

% of SPP #PFs
obtained by instrumenting Leanguard-buddy and Native-buddy

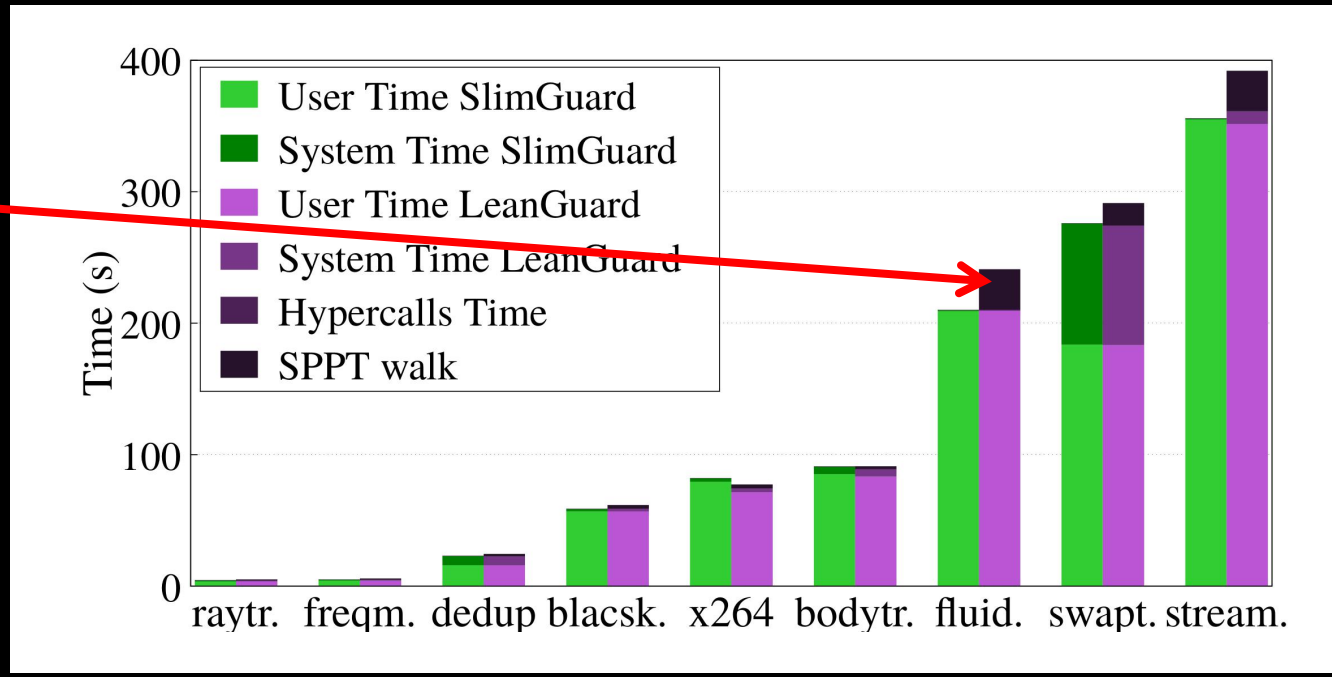
LeanGuard Evaluation: Performance Overhead Results



LeanGuard Evaluation: Performance Overhead Results

The gap between SlimGuard and LeanGuard is essentially the SPPT walk

LeanGuard incurs only ~7.7% overhead on average: can be acceptable to a user whose priority is security



Summary and Future Directions

Summary

1. Introduction to OoH (Out of Hypervisor):
   - New virtualization research axis
   - Exposure of hypervisor-oriented HW virt. features to guest processes
   - Excellent alternative to nested virtualization
2. The large ecosystem of hardware virtualization features can help improving existing sofware-based tools
3. Think of OoH and virt. feature categorization from the conception / design of HW virtualization features

Future Directions

OoH in bare-metal:

✓ Make host userspace applications take advantage of HW virtualization functionalities

OoH for other hypervisors, processors, and architectures:

✓ OoH' principles can be applied to processors other than Intel (e.g., AMD)

✓ OoH can find applicability to AArch64 architectures (e.g., with NEVE)

Materials: OoH is open-sourced and available on Github https://github.com/bstellaceleste/OoH

Publications and Awards

Pulications		Awards
International	National	
Out of Hypervisor (OoH): Efficient Dirty Page Tracking In Userspace Using Hardware Virtualization Features, **S. Bitchebe & A. Tchana**, **SC 2022**	Applying Intel SPP Feature for Buffer Overflows Mitigation, **S. Bitchebe et al.**, **Compas 2022**	2022 **Google** Scholarship for Women in Computer Science
Extending Intel PML for Hardware-Assisted Working Set Size Estimation of VMs, **S. Bitchebe et al.**, **VEE 2021**	Caracterization and Methodology of Buffer Overflow Analysis, Y. KONE, **S. Bitchebe** & A. Tchana, **Compas 2022**	2021 **Microsoft** Research Ph.D. Fellowship
Study of Intel PML Effectiveness, **S. Bitchebe et al.**, **Eurosys Doctoral Workshop 2019**	sQemu: Toward a Scalable Virtual Storage, K. Nguetchouang, **S. Bitchebe** et al., **Compas 2022**	2021 L'**Oréal-UNESCO** For Women in Science Program
Rejects Eurosys 2023, MICRO 2019, JSS 2020, Eurosys 2020, SIGMETRICS 2020, IFIP PERF 2020, ISCA 2022	Hardware Assisted Virtual Machine Page Tracking, **S. Bitchebe et al.**, **Compas 2019**	2021 **NEC Lab** Ph.D. Research Fellowship