EPTI: Efficient Defence against Meltdown Attack for Unpatched VMs

Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang,
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

https://www.usenix.org/conference/atc18/presentation/hua

This paper is included in the Proceedings of the
2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-931971-44-7

EPTI: Efficient Defence against Meltdown Attack for Unpatched VMs

Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, Binyu Zang

Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
{huazhichao123,dd_nirvana,xiayubin,haibochen,byzang}@sjtu.edu.cn

Abstract

The Meltdown vulnerability, which exploits the inherent out-of-order execution in common processors like x86, ARM and PowerPC, has shown to break the fundamental isolation boundary between user and kernel space. This has stimulated a non-trivial patch to modern OS to separate page tables for user space and kernel space, namely, KPTI (kernel page table isolation). While this patch stops kernel memory leakages from rouge user processes, it mandates users to patch their kernels (usually requiring a reboot), and is currently only available on the latest versions of OS kernels. Further, it also introduces non-trivial performance overhead due to page table switching during user/kernel crossings.

In this paper, we present EPTI, an alternative approach to defending against the Meltdown attack for unpatched VMs (virtual machines) in cloud, yet with better performance than KPTI. Specifically, instead of using two guest page tables, we use two EPTs (extended page tables) to isolate user space and kernel space, and unmap all the kernel space in user's EPT to achieve the same effort of KPTI. The switching of EPTs is done through a hardware-support feature called *EPT switching* within guest VMs without hypervisor involvement. Meanwhile, *EPT switching* does not flush TLB since each EPT has its own TLB, which further reduces the overhead. We have implemented our design and evaluated it on Intel Kaby Lake CPU with different versions of Linux kernel. The results show that EPTI only introduces up to 13% overhead, which is around 45% less than KPTI.

1 Introduction

The recently discovered Meltdown [16] and Spectre [14] vulnerabilities allow unauthorized processes to read data of privileged kernel or other processes, which brings severe security threat especially to cloud platforms. Currently, Intel has released micro-code patches to fix the Spectre vulnerability. However, in order to fix the Meltdown vulnerability, which is much more serious and easier to exploit, users are required to apply a kernel patch named KPTI (kernel page table isolation) [30] that uses two page tables to host kernel and user programs to isolate kernel address space from any user process. While this patch can effectively defend the Meltdown attacks, it brings three issues, which leaves thousands of millions of unpatched machines in danger.

First, the patch has to be applied manually by every user. In cloud environment, although the cloud administrators can patch the host OS, they cannot directly patch guest OS running in VMs (virtual machines) since they are not allowed to do so. For example, Amazon "recommend that customers patch their instance operating systems to address process-to-process or process-to-kernel concerns of this issue" [12]. However, many cloud users are not capable of doing such system maintenance.

Second, the patch may depend on specific versions of kernel, especially for Linux. Till now, Linux community just released version 4.15 that contains the patch. The patch may not work on some early versions of kernel like 4.4 [28]. It is expected to take a long time before the patch can be applied to all the versions of Linux kernel.

Third, the patch may incur non-trivial performance slowdown. The KPTI patch makes the kernel and user process use different page tables, which causes TLB-flush during the switching between user-mode and kernel-mode and thus increases the rate of TLB miss. Prior evaluation results show that for some system-call intensive workload, the performance penalty may be high as 30% in VMs [22]; our own experiments confirmed such performance slowdown (Section 6).

In this paper, we present an alternative approach to defending against Meltdown attack for VMs in cloud. Our approach, namely EPTI, can be applied to unpatched guest VMs without users' awareness and can achieve better performance than KPTI at the same time. First, instead of using two gPTs (guest page tables) as in KPTI, EPTI uses two EPTs (extended page tables), namely EPT_k and EPT_u, to run the kernel and user processes, correspondingly. The guest kernel and user still share one gPT, but in user mode, the gPT entries for mapping kernel address space are set to zero in EPT_u, which forbids any translation of address within kernel space to mitigate the Meltdown attack. Second, we leverage one of Intel's hardware features for virtualization, named *EPT switching*, to switch the two EPTs within the VM itself without causing any VMExit. We use binary instrumentation to insert two trampolines at the entrance and exit of guest kernel to do the EPT switching, which does not require kernel's source code and has little (if any) dependence on kernel versions. Third, through a detailed micro-architectural analysis, we find that EPT switching can be more efficient than gPT switching. Since each EPT has its own TLB, when switching the EPTs there will be no TLB flushing by hardware, which is the main reason of performance degradation of KPTI. We also adopt several optimizations to minimize the number of VMExits to further reduce the overhead. Fourth, EPTI can be seamlessly deployed in the cloud by combining with live VM migration [5]: a host can migrate away all the guest VMs, patch the host hypervisor with EPTI, and then migrate all the VMs back.

We have implemented EPTI on KVM and use unmodified Ubuntu distribution as guest VM for evaluation. We conduct a detailed security analysis as well as evaluation to show that our EPTI can achieve the same security guarantee as KPTI. We also evaluate real-world benchmarks to measure the performance overhead. The results show that the *average* performance overhead on server applications of EPTI is about 6%, which is 45% lower than KPTI whose *average* overhead is 11%.

To summarize, this paper makes the following contributions:

- An EPT-level isolation of kernel's and user's address spaces to defend against Meltdown attack for unpatched guest VMs.
- Several optimizations to achieve better performance than the current solution KPTI.
- A prototype of our design on real hardware for performance and security evaluation.

2 Motivation and Background

2.1 Meltdown Attack and KPTI

The Meltdown vulnerability was published in January 2018, known to affect Intel's x86 CPU, ARM Cortex-A75 [16] and some versions of PowerPC processor [11]. Through this attack, a malicious user application can steal contents of kernel memory in two steps. Step-1: to access kernel address A and to leverage its data as an index to access the cache; step-2: to get the data through cache covert channel. The key problem here is that the Step-1 is executed reordered and will be canceled eventually, but the cache layout is affected without rollback. Since the kernel will typically map all the physical memory within its memory space, the malicious application can potentially get all of the memory contents.

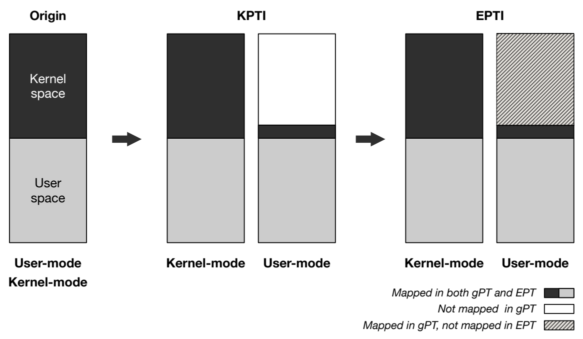


Figure 1: Page table isolation. For a VM, KPTI uses two gPTs and one EPT, while EPTI uses one gPT (since VM is not patched) and two EPTs.

KPTI (kernel page table isolation) [30] is based on KAISER (kernel address isolation to have side-channels efficiently removed) [19], which is proposed to defend against the Meltdown attack. This patch separates user space and kernel space page tables entirely, as shown in Figure 1. The one used by kernel is the same as before, while the one used by application contains a copy of user space and a small set of kernel space mapping with only trampoline code to enter the kernel. Since the data of kernel are no longer mapped in the user space, a malicious application cannot directly de-reference kernel's data address, and thus cannot issue Meltdown attack. KPTI has been merged to the mainstream Linux kernel 4.15, which was released on 28 Jan, 2018. However, the patch still has problems on previous Linux kernel versions. For example, it is reported that some Ubuntu user "just got the Meltdown update to kernel linux-image-4.4.0-108-generic but this does not boot at all" [28]. Considering the patch needs to be applied manually by system administrators, it may take a long time before most of the machines getting the patch deployed.

2.2 Overhead of KPTI

KPTI introduces performance overhead since both entering-kernel and exiting-kernel require additional page table switching. The switching is done by loading the CR3 register, which takes around 300 cycles. Meanwhile, since TLB (translation lookaside buffer) will be flushed during CR3 changing, the performance will further be affected due to higher TLB miss rate. There

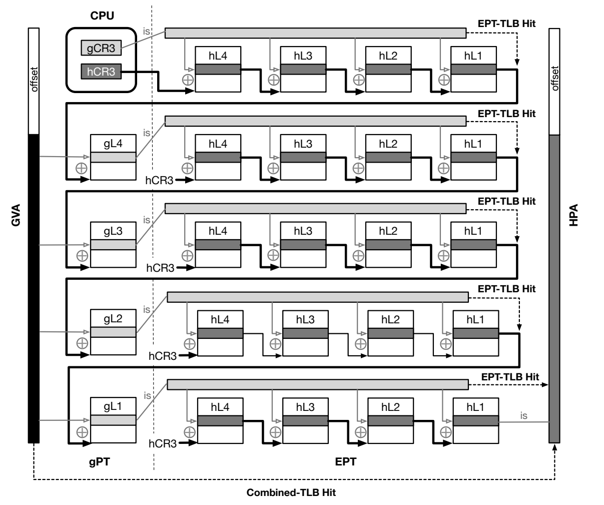


Figure 2: Process of translating GVA to HPA in an x86-64 guest VM. The gCR3 of CPU points to gPT and hCR3 points to EPT.

are many reports on evaluations of KPTI's overhead, which show that KPTI could lead to significant performance cost (up to 30%), particularly in syscall-heavy and interrupt-heavy workloads [30, 25, 18].

On processors that support PCID (process-context identifiers) feature, a TLB flush can be avoided and the performance overhead of KPTI can be reduced. PCID is a 12-bit tag of page table and is saved as a part of a TLB entry. For two page tables with different tags, their TLB entries can co-exist in the CPU and no TLB flush is needed when switching between the two page tables. Existing report shows that after enabling PCID, the overhead of KPTI on PostgreSQL's read-only test on Intel Skylake processor reduces from 17-23% to 7-17% [18]. However, Linux does not support PCID until version 4.14 released on 12 Nov 2017.

2.3 gPT, EPT and TLB

In native environment, PT (page table) is used for translating virtual address to physical address. In virtualization environment, the guest VM (virtual machine), running in non-root mode, only controls its GVA (guest virtual address) to GPA (guest physical address) mapping by gPT (guest page table). The hypervisor, running in root mode, controls each VM's GPA to HPA (host physical address) mapping through a hPT (host page table), which is called EPT (*extended page table*) on Intel platform [1].

Figure 2 shows the procedure of GVA-to-HPA translation on an x86-64 machine with 4 level gPT and EPT. The value of guest CR3 and addresses inside gPT are GPAs, while the value of EPTP and address inside EPT are HPAs. When CPU walks the gPT, it needs to translate all the GPA of needed gPT pages to HPA through EPT.

In order to minimize memory footprint during page walk, the processor has two types of TLB in virtualized environment: EPT-TLB and combined-TLB. The EPT-TLB is used for accelerating translation from GPA to HPA, while the combined-TLB stores entries of translation from GVA to HPA.

[1]The hPT of AMD is called NPT (*nested page table*). Since the Meltdown attack only affects Intel's processor, we use "EPT" instead of "hPT" in the rest of the paper.

2.4 EPTP Switching with *VMFUNC*

VMFUNC is an Intel hardware virtualization extension, which provides VM functions for guest VMs, running in non-root mode, to directly invoke without VMExit. Currently, there is only one VM function provided by *VMFUNC*, named "EPTP switching", which allows software (either in the kernel mode or user mode) in guest VM to load a new EPTP (EPT pointer). Guest can only switch to the EPEP from a list of valid EPTP values configured by the hypervisor. The EPTP switching is supported on all Intel CPUs starting from Haswell architecture.

Performance of EPTP switching: We compare the latency of loading CR3 and EPTP switching. Writing the same value to the CR3 in guest VM costs around 300 cycles, with PCID enabled. While the "EPTP switching" takes about 160 cycles (two different EPTP values, but have the same mappings).

TLB behavior of EPTP switching: We further test the TLB behavior of EPTP switching and find how CPU constructs address mapping in TLB for different EPTs. The operations performed with one EPT will not affect other EPTs, Table 1 shows test results. In the table, "Invalid both EPTs' TLBs then fill EPT-0's TLB" means we first invoke *invlpg* instruction (which is used to flush TLB) in both EPT-0 and EPT-1, and then access the target memory in EPT-0. After that, we access the target memory again in both EPT-0 and EPT-1, and test the access latency. The result means that the EPT-0's is filled while the EPT-1's is not. We also test whether invoking flush TLB operations (write CR3 and *invlpg*) in one EPT will influence the other's TLB or not. We find that both of them flush other EPT's TLB.

3 System Overview

EPTI has three goals:

- **Goal-1**: To achieve the same security level as KPTI.
- **Goal-2**: To support protection on unpatched VMs seamlessly.
- **Goal-3**: To get better performance than KPTI.

Table 1: TLB behaviors of different EPTs during *VMFUNC*.

Action	**Access again in EPT-0**	**Access again in EPT-1**	**Conclusion**
Invalid both EPTs' TLBs then fill EPT-0's TLB	3-5 cycles	120+ cycles	Each EPT has its own mapping in TLB.
Fill both EPTs' TLBs then write CR3 in EPT-0	120+ cycles	120+ cycles	Writing CR3 will flush TLB of all EPTs.
Fill both EPTs' TLBs then *invlpg* in EPT-0	120+ cycles	120+ cycles	*invlpg* will flush TLB of all EPTs.

We first construct two EPTs for each guest VM: EPT_k for kernel and EPT_u for user. The mapping of EPT_k is the same as original EPT (but with different permissions, which will be introduced later), so that the kernel will run just as before. When user applications are running, we need to ensure that they cannot access (even speculatively) any data in the kernel address space.

One intuitive way is to remove all the mappings of HPA of pages used by guest kernel in EPT_u, so that all kernel pages are not mapped when a user process is running. However, this solution does not work since typically Linux kernel will map *the entire GPA* to its GVA space, which is known as *direct mapping*, as shown in Figure 3. It means that we have to remove all of the GPA mappings from EPT, which will also disable the execution of user processes.

Instead, we just remap all the *gPT's pages that map kernel space* in EPT_u to a zeroed page, as shown in Figure 3. Thus, once a user process tries to access kernel address using its GVA, the GVA will never be translated to GPA since the CPU cannot find the corresponding mapping in gPT (refer to the left part of Figure 2). The security guarantee is the same as KPTI (**Goal-1**).

Next, we need to find a way to switch the EPTs at appropriate points. When a user process traps to kernel, the processor should immediately switch to EPT_k by *VMFUNC*. It also switches to EPT_u before the kernel returns to user process. In Linux kernel, there are limited entry points and exit points. The entry points can be located through IDT (interrupt descriptor table) and some specific MSRs (model-specific registers) [2]. The exit points must contain specific instructions (e.g., *sysretq*). Thus, we use *binary instrumentation* to re-write the kernel code on-the-fly to insert two pieces of trampoline code at the entry and exit points, which are mainly used to do the EPT switching. Leverage this method, EPTI can be used together with live migration to seamlessly protect a guest without rebooting it. More details are described in Section 4.3 (**Goal-2**).

In order to unmap the kernel space in EPT_u, we need to track which gPT pages are used for mapping kernel space, and zero them in EPT_u. EPTI tracks the gL3 pages, which are used to translate kernel GVA, and zero them (details in Section 4). We further present our optimizations in Section 5 to reduce the number of VMExits and get better performance (**Goal-3**).

[2]E.g., IA32_LSTAR controls syscall entry.

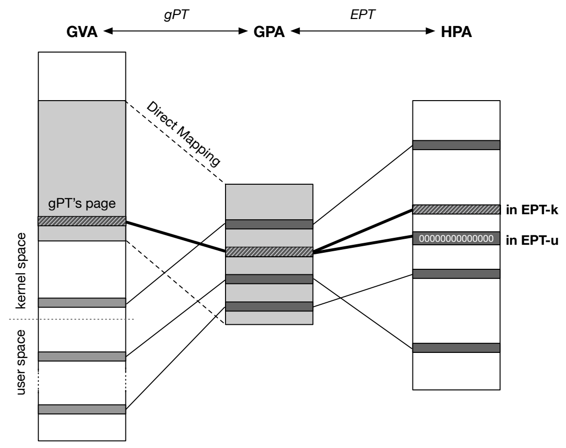


Figure 3: The difference of mapping of EPT_u and EPT_k.

Challenges: There are still many challenges on security and performance in the above design. For example, since it is allowed for a user process to invoke *VMFUNC*, a malicious process may try to switch to EPT_k before issuing Meltdown attack. We will describe our design with more details and present solutions to these challenges in the following text.

4 Design of EPT Isolation

In this section, we introduce the basic design of EPTI. Firstly, we need to construct the EPT_u, which removes all the kernel address mappings. Then, we introduce the basic method of how to track kernel gPT pages and add trampoline code for EPT switching. Finally we construct the EPT_k so that a malicious user cannot switch to it.

4.1 Zeroing gPT for Kernel Space in EPT_u

We remove all the GVA-to-GPA mappings of kernel address space in user mode by zeroing the gPT pages used for address translation in EPT_u. As shown in Figure 3, to zero a gPT page, we remap it to a new zeroed physical page in EPT_u. There are 4 page levels (from gL4 to gL1) in a 64-bit Linux kernel which uses 48-bit virtual address. Since each process has different gL4 pages , to minimize the modification to EPT_u, we only zero the gL3 pages used for kernel address translation (gPT structure is shown in Figure 5).

After zeroing the gL3 pages for kernel space in EPT_u, accessing kernel memory from user mode will trigger a guest page fault since the target GVA is not mapped (although Meltdown attack can bypass permission check, it cannot access non-mapped pages). Since the kernel runs in EPT_k, it can never fill the zeroed gL3 page in EPT_u and the attacker's user process can never access the kernel memory (even speculatively).

4.2 Tracking gPT Pages in EPT_k

In order to zero all the gL3 pages that map kernel space in EPT_u, EPTI first needs to track all the gL3 pages for the kernel. Specifically, by setting the *CR3_LOAD_EXITING* bit in *VMCS* (virtual machine control structure), when a guest kernel changes CR3 it will trap to the hypervisor, which will then walk through the gPT to get a list of all gL3 pages for kernel space mapping. Meanwhile, the guest kernel may allocate new gL3 pages and add them to gL4. In order to track new kernel gL3 pages, all the gL4 pages will be mapped as read-only in EPT_k, so that each time a guest kernel adds a new gL3 page to gL4, it will trap to the hypervisor to update the monitored gL3 page list. We will present some optimization of tracking in Section 5.

4.3 Trampoline for EPT Switching

Listing 1 EPT switching to EPT_k

```
1:  SWITCH_EPT_K:
2:    SAVE_RAX_RCX
3:    movq $0, %rax
4:    movq $0, %rcx
5:    vmfunc
6:    RESTORE_RAX_RCX
```

The trampoline code contains instructions for EPT switching. Listing 1 shows the assembly code for switching from EPT_u to EPT_k. The *%rax* and *%rcx* contain the *VMFUNC index* and arguments passed to the *VMFUNC*. Line 3 means to call the first *VMFUNC* function (EPTP switching, index 0), and line 4 means to switch to EPT-0. Both *%rax* and *%rcx* are caller-saved, so the values need to be saved and restored in the trampoline. The process of *SWITCH_EPT_U* is similar but in the other direction.

Since the trampoline code is used to switch between EPT_k and EPT_u, it needs to be invoked in both EPTs. We need to ensure that: (1) the trampoline is executable in both EPTs and (2) there is a suitable place to store the caller-saved registers.

Mapping trampoline as executable in both EPTs: In EPT_u, only one page with the trampoline code will be mapped in the kernel space. To ensure that, EPTI remaps all the gPT pages (except gL4), which are used to translate the GVA for the trampoline, to new host physical pages. Then all the entries of these pages are set to zero, except those that used for mapping the trampoline (as shown in Figure 4). Entries of the guest IDT and the syscall entry MSR (IA32_LSTAR) will be changed to point to the trampoline code. In EPT_k, EPTI inserts the trampoline code to the end of direct map region of guest kernel, and re-writes the binary of kernel to change the exit points to *jmp* instructions that transfer control to the trampoline.

Saving caller-saved registers: Since *VMFUNC* will not save any register by hardware (which makes it fast), the trampoline code cannot use any register before saving them. One challenge to save these caller-saved registers is to support multi-core. For single CPU core, the value of %rax and %rcx can be saved to a memory page with a fixed address. However, for multi-core, one core may overwrite the saved register values of another core since they write to the same address.

Linux solves this problem by using *gs*-based per_cpu value. During system boot, it allocates a per_cpu memory region for each core. The base addresses of these regions will be recorded through *gs* registers of different cores after entering the kernel (through *swapgs* instruction) [3]. The following access of per_cpu values is performed by *%gs:index*. EPTI cannot leverage this method because: (1) it needs to know some specific semantics of the kernel and (2) it needs to map kernel's per_cpu region into EPT_u.

EPTI provides a per_vCPU memory page to save and restore the caller-saved registers. Specifically, a memory page is mapped into the kernel space in both EPT_u and EPT_k. To enable concurrent accesses from multiple cores, the page will be mapped to different physical pages for different vCPUs, so that when one vCPU saves %rax and %rcx, the values are written to its own page. In our implementation, we modify gPT to map this page to an unused GPA (e.g., the GPA out of the guest's DRAM range). In the EPTs for different vCPUs, we map this GPA to different HPA. In both EPT_k and EPT_u of one vCPU, it is mapped to the same HPA.

Seamless protection: Combined with live migration, EPTI can seamlessly protect a guest VM without rebooting it. The cloud provider can migrate away all the VMs, update the host hypervisor to enable EPTI and migrate all the VMs back. To resume a VM on EPTI, we need to: (1) map the trampoline into the guest; (2) rewrite the entries for interrupts and syscalls (stored in IDT and MSR), as well as the exit points (contain specific instructions e.g., sysretq), to jump to the trampoline; (3) enable the trapping of gPT and guest EPTP switching.

[3]The *swapgs* instruction exchanges the current *gs* value with the value stored in MSR_KERNELGSbase.

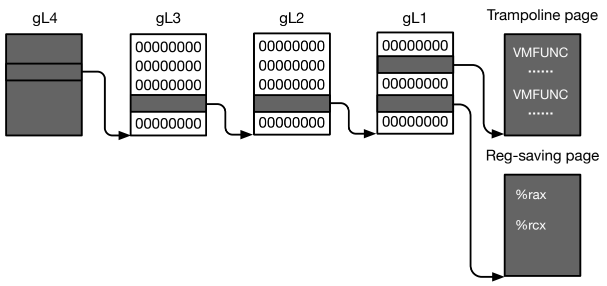


Figure 4: Contents of kernel space of EPT_u, which includes trampoline code page, register saving page, and gPT for mapping these two pages.

4.4 Malicious EPT Switching

The above design relies on an assumption that only the kernel can switch EPT. Unfortunately, the *VMFUNC* instruction can be invoked in either guest kernel mode or guest user mode, which enables an attacker to maliciously switch to EPT_k by *VMFUNC*, issue Meltdown attack and switch back to EPT_u. To defend this attack, EPTI needs to make EPT_k useless for the user process.

By performing real Meltdown attacks, we find that although Meltdown can read the memory without access permission, it cannot fetch code without executable permission even in reorder-execution. Base on this observation, we map all user memory as execute-never in EPT_k. Thus, once the user maliciously switches to EPT_k, all its code will be non-executable.

Specifically, EPTI only maps the guest physical memory (including kernel's code and kernel modules) as executable in EPT_k, and all other guest physical memory is mapped as execute-never. The kernel code is loaded during system booting and will not be changed during execution, EPTI can detect all the corresponding GPAs by searching gPT. The kernel modules are loaded/removed dynamically during runtime, EPTI needs to monitor all the guest physical pages used for them. This is done by trapping all write operations on gPT pages which translates GVA-to-GPA mapping of kernel modules. Since Linux kernel reserves a fixed GVA region for kernel modules, trapping modifications to the corresponding gPT pages will only influence the performance of installing/removing kernel modules.

5 Optimizations

As mentioned in the previous section, EPTI needs to trap both the load-CR3 operation and the write-gL4 in guest VMs. These trapping methods have three performance problems:

- **Useless traps of load-CR3**: EPTI traps guest VM's load-CR3 operations for getting all the gPTs. In fact, EPTI only needs to trap the *new* gPTs, but most of the load-CR3 operations just load old gPTs, which causes a lot of useless VMExits.
- **Access/Dirty bits update**: To trap the modification of a gPT page, EPTI marks it as read-only in EPT_k. However, for each memory access (including read, write and fetch), the CPU will update the A/D bits (access/dirty bits) in the gPT entries which are used for translating the target GVA, even when the A/D bits are already set by previous operations. Thus, whenever the kernel accesses any of its data, it will trigger an EPT violation, which causes a huge number of VMExit.
- **Additional traps of write-gPT**: In Section 4.2, EPTI traps all write-gL4 operations to track all enabled gL3 for kernel space mapping. However, each process has one gL4 page, which means EPTI needs to trap thousands of gL4 pages. Since kernel address mappings are the same for each process, trapping all these gL4 can be optimized.

In this section, we give several optimizations to solve all these performance problems.

5.1 Selectively Tracking Guest CR3

EPTI leverages a hardware feature to reduce the number of VMExit caused by trapping loading old CR3. Intel provides four *CR3_TARGET_VALUE* fields in *VMCS*. A load-CR3 in guest does not cause a VMExit if its source operand matches one of these values. We write the CR3 value, which 1) causes more than *threshold A* VMExits per second or 2) totally causes more than *threshold B* VMExits, to the *CR3_TARGET_VALUE* (A and B can be configured by the VMM).

5.2 Setting gPT Access/Dirty-Bit

In order to eliminate VMExit when CPU setting A/D-bit, we need to allow CPU to write gPT while disallowing kernel to do so. We find that the access path of them are different: the kernel writes gPT through its GVA (using both gPT and EPT), while the CPU writes gPT through its GPA (using EPT only). Thus, EPTI maps gPT pages with write permission in the EPT to allow CPU updating the A/D bits. To trap kernel modifying a gPT page, we redirect the GVA of this page to a new GPA which is mapped as read-only in EPT_k. This is done by (1) modifying the gL1 entry that is used for GVA-to-GPA translation of the target gPT page and remapping the gPT page to a new GPA; (2) mapping the new GPA to the original HPA as read-only, which contains the target gPT page. Thus, only the write access to the gPT page from kernel will trigger a VMExit.

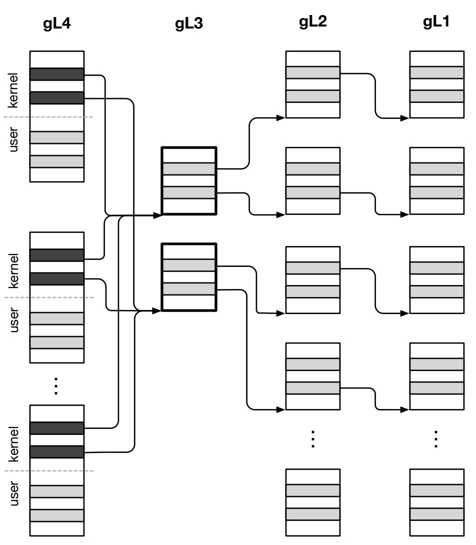


Figure 5: gPT of Linux. The kernel gL3 entries are shared by different gPTs.

5.3 Trapping gL3 Pages Instead of gL4

We adopt another optimization according to the following observations:

- *Most kernel virtual address regions are never changed.* Linux kernel reserve memory regions for different usages [4], and it never changes the mapping of most of these regions (e.g., direct map region is never changed).
- *Each gL3 pages can translate a large virtual space (512GB).* In a guest, it is almost impossible for the kernel to allocate so much virtual memory, so the number of kernel gL3 pages is rarely changed.
- In Linux kernel implementation, kernel only creates a new gL3 page when all entries of existing gL3 pages are in use, or the continuous free entries are not enough.

Based on the above observations, EPTI directly traps the modification of gL3 pages for kernel by default. When the last entry of a gL3 is used, which means the kernel may allocate a new gL3 page later, EPTI starts to trap the load-CR3 and write-gL4 until a new gL3 page is allocated. With this optimization, EPTI almost does not need to trap the operations of load-CR3 and write-gL4, which will reduce most (if not all) of VMExits.

6 Evaluation

In the evaluation, we try to answer these seven questions:

- *Question-1*: Can EPTI prevent Meltdown attacks?
- *Question-2*: How EPTI influences the performance of kernel critical operations (e.g., syscalls)?
- *Question-3*: How EPTI influences the performance of real server applications?
- *Question-4*: How EPTI influences the performance of multiple guest VMs?
- *Question-5*: How many VMExits are reduced by different optimizations of EPTI?
- *Question-6*: Can EPTI work on different kernel versions and how about the performance?
- *Question-7*: Can a guest VM be live migrated to hypervisor with EPTI and what is the performance?

6.1 Evaluation Environment

We do the evaluation on an x86-64 machine with an 8-core Intel Core i7-7700 CPU, 16GB memory and a Samsung 512GB SSD. We implemented EPTI with KVM based on Linux kernel 4.9.75 running in Ubuntu 14.04. We assigned 4 vCPUs (virtual CPUs) and 8GB memory to the guest VM, which runs an Ubuntu 16.04. The Linux kernel 4.9.75 is used as the guest kernel by default. In Section 6.4, we also test the overhead of multiple guest VMs. In Section 6.6, we test various kernel versions in the guest VM.

We isolate four physical cores on the host and each vCPU of the guest is pinned to a physical core. We use *virtio* to virtualize guest disk. During the evaluation, all the clients and server applications are running in the guest VM to reduce the influence of network.

In the performance evaluation, we test five systems: "Linux" (without KPTI), "KPTI" and EPTI with different optimizations, in which "EPTI-No" means EPTI with A/D bits updating, "EPTI-CR3" means applying A/D-bit updating and *CR3_TARGET_VALUE* to reduce VMExit caused by frequently loaded CR3, and "EPTI-CR3+L3" means applying all three optimizations.

6.2 Security Evaluation

First we implemented a PoC (proof of concept) of Meltdown attack, which has three steps: *step-1*: reads secret **S** from *kernel address*; *step-2*: uses **S** as an index to access memory (covert channel); and *step-3*: probes the cache and gets the value of **S**. The PoC also registers a signal handler of the segmentation fault to continuously perform the attack.

We use this PoC to steal *linux_proc_banner*, a value stored in kernel space (the PoC can also steal any other data in the kernel address space). It succeeds on Linux without KPTI, but is failed when using KPTI and EPTI. We then insert a *VMFUNC* in the PoC to make it switch to EPT_k just before step-1. The attack does not work

[4]e.g., In Linux with 48-bit VA, range from 0xffff880000000000 to 0xffffc7ffffffffff is used for direct map, and range from ffffc90000000000 to ffffe8ffffffffff is used for *vmalloc* and *ioremap*.

Table 2: Evaluation results of LMBench, in μs.

Operation	Linux	KPTI	EPTI-No	EPTI-CR3	EPTI-CR3+L3
Null syscall	0.04	0.16	0.12	0.12	0.12
Null I/O	0.07	0.2	0.17	0.17	0.16
Open/Close	0.70	0.93	0.84	0.84	0.83
Signal Handle	0.68	0.81	0.76	0.76	0.76
Fork syscall	72.9	79	80	80	75
Exec syscall	212	243	242	234	221
ctsw 16P/64K	6.07	7.37	7.66	7.66	6.39

on EPTI due to the defense mentioned in Section 4.4. We also try to pass a constant value through the covert channel after switching to EPT_k, which also fails.

The security evaluation shows that our system can successfully defend against existing Meltdown attacks, even if a malicious process switches to EPT_k first. Actually, a user process cannot execute any code in EPT_k. Logically, both EPTI and KPTI isolate the address space of user and kernel mode, so both of them can defend against Meltdown attacks.

6.3 Micro Benchmark

LMBench [21]: To answer *Question-2*, we use LMBench to test the time of some critical operations, e.g., syscall like *fork* and *exec*. As shown in Table 2, the *null* syscalls of unmodified Linux and KPTI take 0.04μs and 0.16μs, respectively. For EPTI, the time is about 0.12μs, which is smaller than KPTI due to the benefit of no-TLB-flushing of *VMFUNC*. The Null I/O, Open/Close and Signal Handle have the similar results as the *null* syscall. In all cases, EPTI performs better than KPTI. There is no difference between EPTI with different optimizations, because these operations do not involve load-CR3 or write-gL4.

For other three operations (*fork*, *exec* and context switch), EPTI-No and EPTI-CR3 take more time than EPTI-CR3+L3 because these operations contain many load-CR3 and write-gL4 operations. In LMBench, the EPTI-CR3 (optimized with *CR3_TARGET_VALUE*) has the same result with EPTI-No since a process is terminated before being identified as *trapping frequently*. The result of LMBench shows that both EPTI and KPTI have overhead on single critical operation, and the overhead of EPTI is smaller than KPTI.

SPEC_CPU 2006 [8]: We evaluate *all* of SPEC_CPU 2006 INT applications under five systems. As shown in Figure 6 (a), there is almost no overhead in both KPTI and EPTI, since these CPU-related applications rarely interact with the kernel.

6.4 Application Benchmark

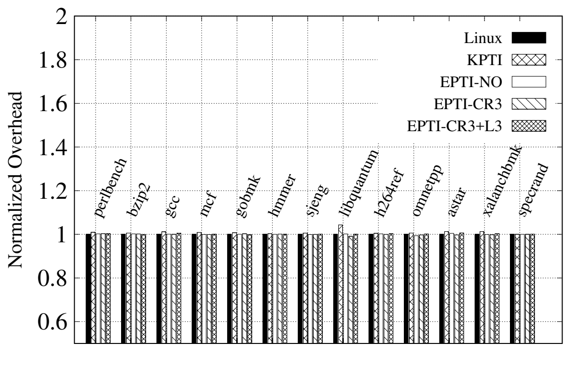
To answer that how EPTI influences the performance of server applications (*Question-3*), we evaluate the performance overhead of file system operations, databases and web servers.

Fs_mark [27] is used for evaluating file system performance. We configure it to continuously create 1MB files in the guest VM and use synchronization method 1 (call *fsync* before close a file), with different number of threads (each thread create 1000 files). The result is shown in Figure 7 (a), the KPTI has 6.5% overhead in single thread while our system has 1.1%. The overhead of both EPTI and KPTI are small because of the slow disk I/O performance for the guest.

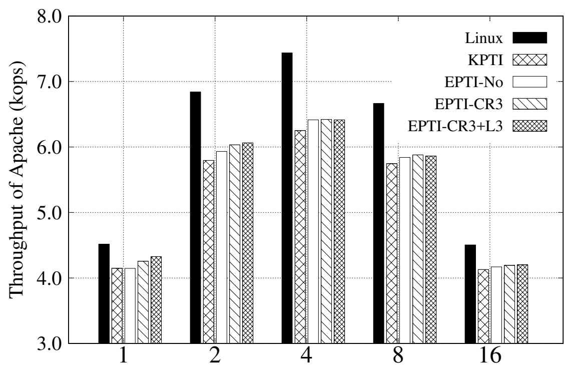
Redis [24] is used for evaluating key-value store workloads. We use the standard redis-benchmark to test the throughput of *SET* and *GET* operation of Redis. The redis-benchmark is configured to use different numbers of threads (from 1 to 8) and the Redis runs with default configuration. Figure 7 (b) shows the result. The X-axis means the test operation and the number of threads used by the client (e.g., *SET-1* means *SET* operation with one thread). On the average, KPTI has about 12% of performance overhead while EPTI has 6%. In the worst case, KPTI has more than 20% overhead and our system has 12%.

PostgreSQL [23] is a relational database. We test its performance with the *pgbench* (a benchmark provided by PostgreSQL based on TPC-B). We test the throughput of read-only and read-write transactions of PostgreSQL under three different loads: *single thread (S)*: using one database client; *normal (N)*: opening 4 test threads and 16 database clients; *heavy (H)*: opening 8 test threads and 64 database clients. The *pgbench* operates on a small database table with 1000 records. Each test is performed on a cleaned table and lasts for 60 seconds. The PostgreSQL is running with default configuration. The result is shown in Figure 7 (c). The unit of throughput of RO transaction is *kops* and the unit of RW is *ops*. Both KPTI and EPTI have small overhead for single thread *pgbench*. The overhead increases dramatically in the normal and heavy loads. In the Heavy-RO test, KPTI has about 12% overhead while our system has about 4%.

MongoDB [3] is a widely-used non-relational database. We use YSCB benchmark to test the performance of it with different workloads. Each workload is performed on a table with 10,000 records and we configure YCSB to use 32 test threads. The MongoDB uses the default configuration. We test all the standard workloads of YSCB (from workload-A to workload-F) and the result is shown in Figure 8 (a). On average, KPTI has about 7% performance overhead while our system has about 2%.

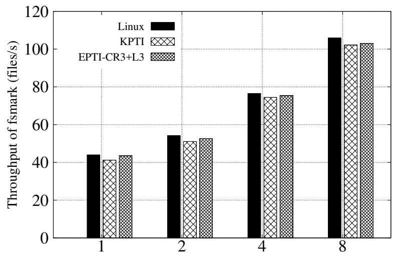


(a) SPEC_CPU INT

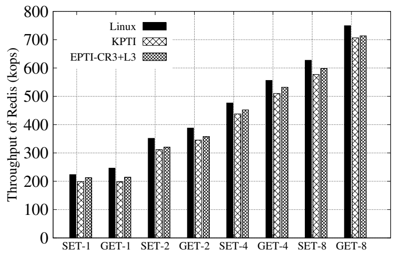


(b) Apache

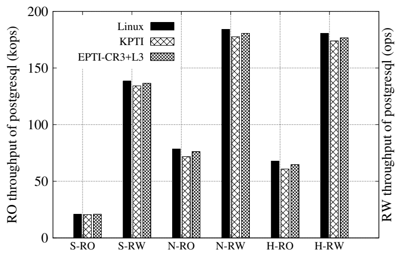
Figure 6: Figure (a) shows the overhead of all INT applications in SPEC_CPU 2006 benchmark, lower the better. Figure (b) shows the throughput of Apache with different number of clients, higher the better.



(a) fs_mark



(b) Redis



(c) PostgreSQL

Figure 7: Figure (a) shows the throughput of fs_mark with different threads. Figure (b) shows the throughput of SET and GET operations of Redis with different threads used by the client. Figure (c) shows the throughput of PostgreSQL under different workloads of RO (read-only) and RW (read-write) transactions. Higher the better.

Apache [1] is a widely-used web server. We use *ab* (apache benchmark) to test the throughput of Apache. It continuously downloads a 1MB static web page from the Apache, with different client threads (1 to 16). The Apache server uses default configuration (event mode). Figure 6 (b) shows the throughput of Apache. The performance drops after 4 client threads since we use 4 vCPUs in the VM. The overhead of KPTI is 15%-18%, while our system (EPTI-CR3+L3) has about 10% overhead.

Nginx [2] is a lightweight web server. We also test it by *ab* benchmark with a 1MB static web page and different threads (1 to 16). The Nginx server runs with default configuration. The throughput of Nginx is shown in Figure 8 (b). In the worst case, the overhead of KPTI is 18% and ours is 12%.

Multiple guest VMs: We evaluate the overhead of EPTI on multiple guest VMs (for *Question-4*). Each VM is configured to have 1 vCPU and 1GB memory, and all the VMs' vCPUs are pinned on 4 physical cores. We use linux 4.15 as the guest kernel, and run a Nginx server as well as an *ab* benchmark tool in each VM. The result is shown in Figure 9, the average overhead of KPTI is about 16% while our system is about 9%

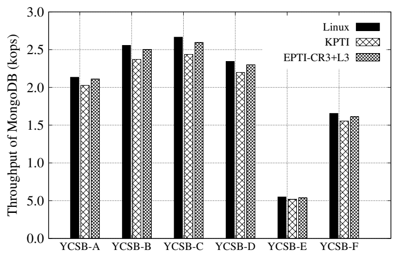
Table 3: Number of VMExit caused by EPTI.

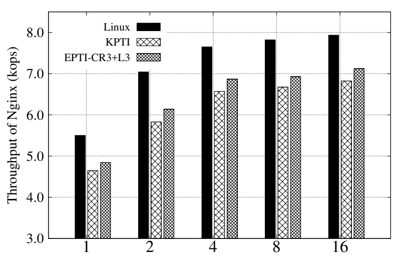
Benchmark	EPTI-No	EPTI-CR3	EPTI-CR3+L3
Redis 1-thread	540	464	0
Redis 8-thread	385	315	0
Apache 4-thread	45406	225	0
Apache 32-thread	40149	623	0
Compile Kernel -j8	609659	551023	0

6.5 Breakdown of Optimizations

To answer how different optimizations reduce the number of VMExit of EPTI (*Question-5*), we test the performance of Apache on EPTI with different optimizations. Figure 6 (b) shows the result. In the best case (1 client thread), EPTI-No has about 9% performance overhead which is almost same as KPTI. EPTI-CR3 only has 5% overhead while EPTI-CR3+L3 has 4%.

To give a detailed breakdown of the performance improvement, we analyze the number of VMExits in EPTI with different optimizations. We calculate the total number of VMExits caused by EPTI of the whole guest in three scenarios: (1) running redis-benchmark to test Redis (1,000,000 operations with 1 or 8 threads); (2) running ab to download 1,000 1MB web pages (with 4 or


(a) MongoDB


(b) Nginx

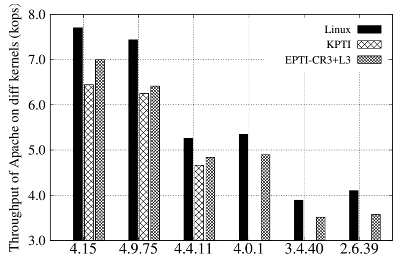

(c) Apache on different kernel versions

Figure 8: Figure (a) and (b) show the throughput of MongoDB and Nginx, higher the better. For MongoDB, we test it with YSCB and X-axis means the different YSCB workloads. For Nginx, we test it by using ab benchmark with different threads. Figure (c) shows the throughput of Apache on different kernel versions, X-axis means the kernel version.

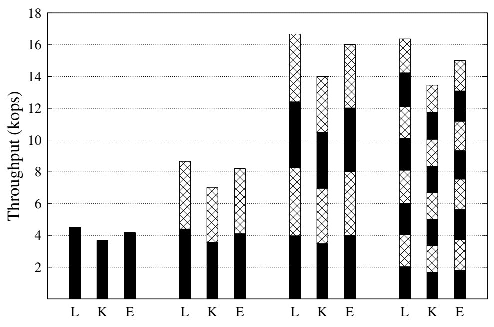


Figure 9: Throughput of Nginx on multiple guest VMs. (L means Linux, K means KPTI, E means EPTI-CR3+L3)

32 client threads); and (3) compiling Linux kernel 4.9.75 with "defconfig" (including kernel modules, "make -j8"). The result is shown in Table 3.

As shown in the table, the optimization of selectively trapping load-CR3 does not have much effect on Redis and kernel compilation, but is effective for Apache. This is because EPTI-CR3 can only reduce the VMExit caused by frequently loading CR3 value, while both Redis and redis-benchmark are single-process that have very few load-CR3 or write-gL4 operations. In kernel compilation, the Makefile creates a gcc process to compile each C file, which produces a huge number of processes with different CR3. Since each gcc process works for a short time, there is no long-term frequently-used CR3 which means the EPTI-CR3 cannot effectively reduce the number of VMExits (theoretically, the result of EPTI-CR3 can be improved by a better algorithm for replacing the value of *CR3_TARGET_VALUE*). On the contrary, Apache with event mode uses a few (typically 4) child processes to manage all the worker threads. Most of the VMExits are caused by loading the CR3 of Apache's child process, which can be optimized by storing their CR3 in *CR3_TARGET_VALUE*.

For all scenarios, there is no modification on kernel gL3, so the number of VMExit can be further reduced to 0 by EPTI-CR3+L3. The reason we still need both optimizations is that operation on kernel gL3 is highly OS dependent, while the optimization of selectively trapping CR3 is more general.

Table 4: VM live migration to host with EPTI, in *ms*.

	KVM w/o EPTI	KVM w/ EPTI
Total time	15779.5 ±1112.03	15782.6 ±1111.86
Downtime	6.1 ±0.82	9.2 ±1.03

6.6 Different Kernel Versions

To answer *Question-6* (could EPTI works on different kernel versions and how about their performance?), we test the performance of EPTI on different Linux kernel versions (selected from 2.6 to 4.15). We run Apache on them and use *ab* benchmark with 4 client threads to evaluate the throughput. The result is shown in Figure 8 (c). Our system has higher performance than KPTI in all the kernel versions (excluding kernels which do not have KPTI support). In the newest kernel 4.15, which enabled PCID, the performance of Linux w/o KPTI is improved obviously. However the KPTI of Linux 4.15 still has about 17% overhead, while our system has 10%.

6.7 VM Migration

To answer the last question, we test the total time and downtime of VM live migration, from a host without EPTI to one with EPTI. The source machine deploys an unmodified Linux kernel 4.9.75 with the same hardware configuration as mentioned, and the target is the one we use in previous evaluation. We use both the unmodified KVM and KVM with EPTI as the target hypervisor. A guest VM can be seamlessly migrated to a hypervisor with EPTI and the overhead is small. Table 4 shows the average migration time together with the stddev (test for 4 times). The downtime increases 3 ms which is caused by the scanning of code region in memory, preparing for two EPTs and binary writing.

7 Related Work

Besides the work mentioned, we now present the systems that also leverage similar hardware features for enhancing system security or performance.

KAISER [19] was proposed to defend against attacks on KASLR [10, 7, 13], which can also prevent Meltdown since it ensures no valid mapping to kernel space in user mode. It is later replaced by KPTI [30] and is merged to Linux kernel from version 4.15.

SecVisor [26] ensures lifetime kernel integrity via setting access permissions in NPT (Nested Page Table, from AMD, similar to Intel's EPT). TrustVisor [20] uses NPT to isolate memory regions used by a security task. CloudVisor [31] de-privileges the hypervisor to non-root mode and uses a separated EPT to host it. Thus, the hypervisor is isolated from guest VMs and is removed from the TCB (trusted computing base). InkTag [9] uses EPT to isolate the address space of a process. SeCage [17] leverages *VMFUNC* to provide two isolated execution environments, one for running security-critical code and the other for the normal code, to defend against attacks like heartbleed [29]. Similarly, MemSentry [15] creates domains (VMs) to hide secret data and uses *VMFUNC* to switch between different domains.

8 Discussion

Supporting x86-32: To support 32-bit linux, EPTI needs two steps, 1) trapping and modifying the gPT and 2) inserting the trampoline. The existing design can be used to trap and construct gPT for 32-bit linux. To add the trampoline, EPTI requires 8KB virtual address region within guest VM which should not be occupied by the VM itself. We could use technology like *shadow IDT* of ELI [6], which leverages extra pages of devices PCI BAR (base address register) in guest VM to insert additional pages.

Supporting five-level page table: 64-bit Linux also provides five-level page table (the root gPT is gL5). EPTI can trap all enabled gL4 pages and zero them in EPT_u to perform the user-kernel isolation. All the trap and zero methods are same as the four-level page table.

Supporting Windows: Technically, the design of EPTI can be applied to Windows or other OSes. All the kernel entries of Windows kernel can be detected by trapping the modification of IDT and MSRs, so that a trampoline can further be added. After that, EPTI can construct the EPT_u and EPT_k after knowing the virtual memory layout of Windows kernel.

Transparency to guest VMs: EPTI modifies the code and gPT of the guest. In current implementation, these modifications are not transparent to the guest VM. These modifications will not affect functionalities like VMI (virtual machine introspection) and kernel integrity check. For the VMI case, we keep the original address mapping with only different permission so the address translation in VMI can be done as before. For the kernel integrity check case, we do not change existing kernel code, so that its hash value will not be changed. Moreover, the VMM can make the modifications transparent to the guest. Features like *XnR* (execute-no-read) [4] can be used to prevent kernel from reading the trampoline code page while still allowing to execute the code, and the access to the modified gPT pages can also be trapped.

Compared with KPTI: EPTI has three advantages compared with KPTI: compatibility, performance and seamless deployment. Even when the KPTI is patched on all Linux versions, EPTI is still valuable for its low performance overhead and seamless deployment without guest rebooting.

9 Conclusion

The publish of Meltdown vulnerability makes public servers in danger, especially those in cloud. The KPTI solution requires server owners to apply the patch manually, which currently supports only a few of kernel versions and may introduce non-negligible performance overhead. This paper presents EPTI, a new solution to Meltdown vulnerability that can be applied to unpatched VMs and with less overhead. Specifically, our solution uses two EPTs (extended page tables) to isolate user space and kernel space, and unmaps all the kernel space in user's EPT to achieve the same effort of KPTI. EPTI leverages two hardware features to reduce the performance overhead: first, the switching of EPTs is done through a hardware-support feature called *EPTP switching* within guest VMs without hypervisor involvement. Second, *EPTP switching* does not flush TLB since each EPT has its own TLB, which further reduces the overhead. By leveraging live migration, EPTI can seamlessly protect a guest VM without rebooting it. We have implemented our design and evaluated on Intel Kaby Lake CPU with different versions of Linux kernel. The results show that EPTI only introduces up to 13% overhead, which is around 45% less than KPTI.

Acknowledgments

This work was supported by the National Key Research and Development Program of China under Grant No. 2016YFB1000104, the National Natural Science Foundation of China under Grant Nos. 61572314 and 61525204.

References

[1] Apache http server. https://www.apache.org/. Referenced Feb 2018.

[2] Apache http server. https://nginx.org/cn/. Referenced Feb 2018.

[3] Mongodb. https://www.mongodb.com/. Referenced Feb 2018.

[4] BACKES, M., HOLZ, T., KOLLENDA, B., KOPPE, P., NÜRNBERGER, S., AND PEWNY, J. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), ACM, pp. 1342–1353.

[5] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2* (2005), USENIX Association, pp. 273–286.

[6] GORDON, A., AMIT, N., HAR'EL, N., BEN-YEHUDA, M., LANDAU, A., SCHUSTER, A., AND TSAFRIR, D. Eli: bare-metal performance for i/o virtualization. *ACM SIGPLAN Notices 47*, 4 (2012), 411–422.

[7] GRUSS, D., MAURICE, C., FOGH, A., LIPP, M., AND MANGARD, S. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security* (2016), ACM, pp. 368–379.

[8] HENNING, J. L. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News 34*, 4 (2006), 1–17.

[9] HOFMANN, O. S., KIM, S., DUNN, A. M., LEE, M. Z., AND WITCHEL, E. Inktag: Secure applications on an untrusted operating system. In *ACM SIGARCH Computer Architecture News* (2013), vol. 41, ACM, pp. 265–278.

[10] HUND, R., WILLEMS, C., AND HOLZ, T. Practical timing side channel attacks against kernel space aslr. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE, pp. 191–205.

[11] IBM. Potential impact on processors in the power family. https://www.ibm.com/blogs/psirt/potential-impact-processors-power-family/. Referenced Jan 2018.

[12] INC., A. Processor speculative execution research disclosure. https://aws.amazon.com/cn/security/security-bulletins/AWS-2018-013/. Referenced Jan 2018.

[13] JANG, Y., LEE, S., AND KIM, T. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 380–392.

[14] KOCHER, P., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints* (Jan. 2018).

[15] KONING, K., CHEN, X., BOS, H., GIUFFRIDA, C., AND ATHANASOPOULOS, E. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems* (2017), ACM, pp. 437–452.

[16] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown. *ArXiv e-prints* (Jan. 2018).

[17] LIU, Y., ZHOU, T., CHEN, K., CHEN, H., AND XIA, Y. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 1607–1619.

[18] MAILING LIST, P. heads up: Fix for intel hardware bug will lead to performance regressions. https://www.postgresql.org/message-id/20180102222354.qikjmf7dvnjgbkxe@alap3.anarazel.de. Referenced Jan 2018.

[19] MAURICE, C., AND MANGARD, S. Kaslr is dead: Long live kaslr. In *Engineering Secure Software and Systems: 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings* (2017), vol. 10379, Springer, p. 161.

[20] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., AND PERRIG, A. Trustvisor: Efficient tcb reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 143–158.

[21] MCVOY, L. W., STAELIN, C., ET AL. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference* (1996), San Diego, CA, USA, pp. 279–294.

[22] NEWS, H. Kpti overhead of redis. https://news.ycombinator.com/item?id=16079457. Referenced Jan 2018.

[23] POSTGRESQL. Postgresql database. https://www.postgresql.org. Referenced Feb 2018.

[24] REDIS. Redis database. https://redis.io. Referenced Feb 2018.

[25] REGISTER, T. Kernel-memory-leaking intel processor design flaw forces linux, windows redesign. https://www.theregister.co.uk/2018/01/02/intel_cpu_design_flaw/. Referenced Jan 2018.

[26] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 335–350.

[27] SOURCEFORGE. Fs_mark. https://sourceforge.net/p/fsmark/wiki/Home/. Referenced Feb 2018.

[28] UBUNTU. Meltdown update kernel does not boot. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1742323. Referenced Jan 2018.

[29] WIKIPEDIA. Heartbleed. https://en.wikipedia.org/wiki/Heartbleed. Referenced Jan 2018.

[30] WIKIPEDIA. Kernel page-table isolation. https://en.wikipedia.org/wiki/Kernel_page-table_isolation. Referenced Jan 2018.

[31] ZHANG, F., CHEN, J., CHEN, H., AND ZANG, B. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 203–216.