Richard Fellner, BSc

A Countermeasure against Kernel Virtual Memory Side-Channel Attacks

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Daniel Gruß, BSc

Institute for Applied Information Processing and Communications

Graz, September 2020

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date, Signature

Abstract

The Kernel of Operating Systems is a nowadays a well-liked target of cybercriminals. Many attack schemes they perform make use of hardware-depending side-channels timing leaks to obtain information of the kernel virtual memory layout upon runtime. This information may then be used to break Kernel Address Space Layout Randomization (KASLR) and perform follow-up attacks. This thesis introduces a way to prevent such attacks. Therefore, we extend the operating system kernel, a new countermeasure: Kernel Address Isolation to have Side-channels Efficiently Removed (KAISER). KAISER isolates the Kernel Virtual Memory against side-channel attacks from the User Space. It creates virtual shadow mappings, so most of the kernel is unmapped in User Mode, and the memory caches are invalidated upon context switch. This way, malicious user programs are not able to gather any information about the kernel virtual memory any longer. The implementation runs under the Linux kernel on x64_64 platforms. Test results for different Intel CPUs show that most attacks can be countered, with a performance impact of down to 0.28% on up-to-date Intel CPUs.

Keywords: Linux kernel, Side-Channel Countermeasure, x86_64, Kernel Virtual Memory Isolation

Kurzfassung

Der Betriebssystem-Kernel ist ein beliebtes Ziel von Cyber-Kriminellen. Sie verwenden zeitbasierte Seitenkanalangriffe um Informationen vom Speicherlayout des Kernels zu erhalten, und die Kernel Address Space Layout Randomization (KASLR) zu brechen. Diese Informationen können dann für einen Folgeangriff verwendet werden. In dieser Arbeit wird eine Methode vorgestellt, die diese Art von Angriffen verhindert. Dafür erweitern wir den Betriebssystem-Kernel um einem Patch: Kernel Address Isolation to have Side-channels Efficiently Removed (KAISER). KAISER isoliert den virtuellen Kernel-Speicher gegen Seitenkanalangriffe von Benutzer-Programmen. Hierzu erstellt KAISER eine Schattenkopie des Speicherlayouts, und entfernt den Kernel-Speicher im Benutzermodus. Des Weiteren invalidiert es bei Kontextwechseln Zwischenspeicher des virtuellen Speichers, die für Angriffe genutzt werden können. Durch diese Gegenmaßnahmen können böswillige Benutzerprogramme keine Informationen über das Speicherlayout des Kernels mehr erhalten. KAISER ist in dem Linux Kernel für x86_64 CPUs eingebaut. Tests zeigen, dass viele Angriffe durch diesen Patch verhindert werden, und die Arbeitsleistung Einbußen von nur 0.28% auf modernen Intel CPUs erfährt.

Stichwörter: Linux Kernel, Seitenkanal-Gegenmaßnahmen, x86_64, Speicherisolation

Acknowledgements

First of all, I would like to thank my supervisor Daniel Gruss for the excellent assistance. He supported me in all phases of my master thesis, and I would not be where I am today without him. In general, I want to thank all people at the IAIK for their great support. I especially thank Michael Schwarz and Moritz Lipp for sharing their broad theoretical and practical expertise with me. And I also thank Stefan Mangard, my prior supervisor, to enable me to take this thesis topic and support me.

I want to thank all colleagues and friends from university, especially Philipp, Lukas, Stefan, Clemens, Irfan, Karl, Samuel, Christiane, David, Philipp, Tomislav, and Stephan for the joyful study time at Graz University of Technology.

My girlfriend Seyda, my parents Alexandra and Herbert, my brother Konrad, and all my close friends supported me throughout my master thesis. I am delighted that I have you in my life, and I am thankful for all the mental support.

Last but not least, I want to thank all my colleagues from ams AG. They gave me the opportunity to finish my master's thesis. Without their assistance, I may have never been able to finish it.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Operating systems have always been a popular target for attackers. They are running on most desktop PCs, servers, smartphones, tablets, and are becoming more common on embedded systems. Attackers with user privileges try to obtain information from the kernel. To prevent attacks on the operating system kernel, many countermeasures have been developed over the past years. Modern operating systems are protected against user access by using hardware memory protection features. Operating systems use privilege modes in combination with paging to prevent user programs from attacking the kernel. Supervisor Mode Access Protection (SMAP) and Supervisor Mode Execution Protection (SMEP), and are used to block kernel threads from unintentionally executing user code or accessing user data. Also, modern operating systems use KASLR to impede attacks that need knowledge about the kernel memory layout to be successful.

Still, attackers find new ways to work around such countermeasures. CPU features used to boost the program performance, often leak information about other processes or the kernel as a side effect. These leaks can be used to obtain secret information about the operating system kernel at run-time. Therefore, randomizing the kernel address space layout is not an effective countermeasure any longer. The layout is only shuffled upon kernel boot and module load time but it is not changed afterward. Attackers can recover the memory layout at run-time and bypass the Kernel Address Space Layout Randomization. There is no efficient method to fix these software-based micro-architectural side-channel attacks yet. So, they are still a serious security issue, which is present on most servers and desktop computers.

Problem Statement Gruss et al. [28], Hund et al. [37], and Evtyushkin et al.

[20] presented new attacks to obtain information of the kernel memory layout at run-time. This way, they can break the KASLR. They use the fact that on many processor architectures, user programs and kernel share the virtual address space. The gained information is used to obtain secret data or to assist other attacks. When the CPU resolves a virtual address, it leaks timing information about the mapping level and cache status. These attacks can be mitigated in hardware, but doing so lowers the performance of the processors. So these side-channel attacks will unlikely be fixed in the hardware. Therefore, an efficient way of how to fix the side channels in software has to be found.

Approach In this thesis, we present a way to isolate the kernel against attacks from user programs. The main contribution of the thesis is a technique called KAISER. We present a prototypical implementation for the Linux kernel on x86_64-processor-architectures. KAISER removes the kernel space from the virtual memory mapping while the system is in user mode, and clears all Translation Lookaside Buffers (TLBs) and paging structure caches. By doing so, malicious user programs are not able to obtain information about the virtual address space of the kernel any longer. To do so, a shadow mapping of the virtual memory is created for each process. When a thread is in kernel mode, the entire memory mapping is mapped. In user mode, only a small part of the kernel memory is mapped, to minimize the attack space. This small mapping is needed for virtual memory structures used by hardware features, and to switch between user- and kernel mode.

1.1 Motivation

Since the thesis presents a new countermeasure, several research questions need to be clarified. Which attacks on the kernel virtual address space can be mitigated in software? Is additional hardware support needed to fix the side channels?

The thesis also addresses engineering questions. What is the best way to implement the countermeasure on the Linux kernel? How effective are the software mitigations? Is there remaining leakage? Does the implementation introduce new side channels, which can be used by attackers? What is the performance impact on the system?

This master thesis answers these questions. We evaluate KAISER against previously known attacks, and detail the remaining attack surface. We also approximate its run-time and memory overhead. Furthermore, we discuss other countermea-

sures against side channel attacks on Kernel Page Table Isolation (KPTI). We present benchmarks of the KAISER run-time overhead against a kernel without KAISER. In the end, we present new attacks that KAISER mitigates, and discuss the real world impact of KAISER.

Outline This thesis is structured as follows.

Chapter 2 presents the background information, which is required to get a basic understanding of the attacks and the solution to prevent them. It starts with an overview of CPU architectures, their virtual memory support features, and some mechanisms used by operating systems. We discuss CPU protection features as well as the CPU mechanisms that create the side channels. We then describe the main principles of modern operating systems, and focus on the Linux kernel entry handlers and memory layout. Then, we discuss a set of timing side-channel attacks on KASLR.

After the background, chapter 3 explains how the kernel memory is hardened against side-channel attacks. It shows how the shadow memory is structured, and how the kernel-user switch and the virtual memory mapping has to be adapted. The strengths and weaknesses of the countermeasure are compared and evaluated.

In chapter 4, we detail the design and implementation of KAISER in the Linux kernel. We show how the existing kernel is modified. Then, we introduce the virtual shadow memory. KAISER uses architectural tricks to lower the impact on the system. We present alternatives to protect the kernel virtual memory layout against side channel attacks, and compare them against KAISER.

To confirm that the introduced countermeasure works, we evaluate the security of KAISER in chapter 5. We evaluate the efficacy of an unmodified kernels as well as a kernel with KAISER.

The results of this master thesis are broadly used in practice. We emphasize that only a very small percentage of Laptops, Desktops, and Servers do not use the technique developed in this master thesis in practice today. In Chapter chapter 6, we present the successor of KAISER, KPTI, are evaluated in terms of memory and computational overhead. Then, we discuss new attacks on the kernel virtual memory. One of these attacks is Meltdown. We show that KAISER prevents Meltdown attacks. In the end, we discuss the impact of Meltdown and this thesis in a broader context on different systems.

Chapter 7 summarizes and concludes this master thesis.

Chapter 2

Background

This chapter provides the background that this master thesis builds upon. It shows the need for a countermeasure to mitigate microarchitectural side-channel attacks on modern operating systems. The first section discusses CPU features. It introduces the reader to hardware mechanisms for operating systems and security mechanisms. It also gives a detailed explanation of virtual address resolution on the x86_64 architecture. The next section provides an overview of the Linux kernel. It starts with an overview of the most commonly used operating systems. Next, it explains the Linux kernel structure and its address space layout. We provide an overview of essential parts of the kernel. The next section contains information about state-of-the-art kernel memory protection principles. It explains the essential Linux countermeasures against memory-based attacks. At last, recently discovered attacks on operating system kernels are presented. These attacks use hardware side channels to obtain information about the kernel address space from user programs. Before this work, there were no countermeasures to fix the hardware weaknesses. This thesis presents a possible solution to make such side-channel based attacks impossible. The attacks are used in Chapter 5 to confirm the effectiveness of the presented countermeasure.

2.1 CPU Features

A **CPU** is a computing unit used to execute programs. It contains one or more cores. Cores support an instruction set to perform operations, manipulate data, and communicate with other components. Registers are used to represent the internal state of the CPU. The CPU provides peripheral interfaces to communicate with external or internal devices. Programs and data are persistently stored on an

external data storage device. This device is usually a Hard Disk Drive (HDD) or Solid State Drive (SSD). For computation, a CPU uses external Random Access Memory (RAM). The RAM is non-persistent, *i.e.*, it is erased on every power loss. The reason why RAM is used is that it is faster than persistent data storage devices. When physical memory is mentioned in this thesis, we mean RAM. The CPU can directly or indirectly read from or write to memory.

Program Execution. To execute a program, it loads data from the storage into memory. Program memory consists of different virtual memory sections. When the program is executed, it is mapped into the physical memory. The loading can also be done lazily. Parts of the sections are then only loaded and copied into memory when they are required first. The most common sections are as follows [50, ch. 6.3]:

- The `.text` section contains the executable code of a program.
- The `.data` section contains the initialized static data.
- The `.rodata` section contains the read-only static data.
- The `.bss` section contains the uninitialized static data.
- The stack stores temporary data. The temporary data size is unknown at compile-time, and therefore, the stack is dynamically allocated. It contains local variables and stores information where to return from a function.
- The heap provides space for dynamic program data, which cannot be allocated at compile-time but cannot be placed on the stack. The heap allows allocating data chunks, which then can be used by the program. The program frees the heap chunks after they are not required any longer. The heap grows upwards (from lower to higher addresses).

There are many types of CPUs available. They differ in their instruction set, the cores, registers, the supported peripherals, and hardware features. Two of the most widely spread Instruction Set Architectures (ISAs) are x86 and Advanced RISC Machine (ARM) CPUs.

Multiprocessor Systems. One way to speed up the system's performance is to use multiple cores instead of one central core for computation. These systems are called multi-processor CPUs. The cores share their physical memory and Input/Output (IO) but have their general registers and specific control registers and status registers. This way, multiple programs can run in parallel for Symmetric Multiprocessing (SMP) instead of scheduling them sequentially.

The x86 architecture is a Complex Instruction Set Computing (CISC) ISA initially designed by Intel [49]. Nowadays, there are various x86 sub-architectures for 16 bit, 32 bit, and 64 bit. Most personal computers and servers use such CPUs. IDC [98] estimated that 12.2 of 14.6 billion dollars are spent yearly on x86 server CPUs in the fourth quarter of 2016. This thesis only focuses on 64-bit implementations. For AMD, they are called x86_64 or AMD64. The equivalent of Intel is Intel 64 (not IA-64, which is the Intel Itanium architecture). The x86 architectures are backward-compatible, which means an x86_64 CPU can also run a 16-bit or 32-bit program. The 16-bit mode is called Real Mode, the 32-bit mode is called Protected Mode, and the 64-bit mode is called Long Mode. All x86 CPUs boot into real mode first and switch to the protected mode or long mode afterward. x86_64 CPUs also support other modes. These other modes are not relevant to this thesis. Thus we omit them in this thesis for the sake of brevity.

ARM also developed multiple, widely used ISAs. ARM CPUs are mostly used for mobile, home, enterprise, and embedded devices. In 2015, 95 % of the smartphones were sold with an ARM processor [58]. ARM offers different ISAs for 16-bit and 32-bit Reduced Instruction Set Computing (RISC) instructions and 32-bit and 64-bit general-purpose registers. In this master thesis, we focus on x86 64-bit systems. ARM architecture features are just briefly mentioned for context.

Memory Management . A CPU requires to manage the physical memory for several reasons. The address space layout of the memory does not have to be linear. The address space can have holes that do not map to RAM. Every memory address can be read, written, or executed all the time. The memory available is usually not known until the system boots. However, a program requires to have a defined address space layout. Otherwise, it could not jump or branch to a specific address or access data. Even worse, two independent programs with overlapping address space layouts could not run at the same time. One way to handle these problems is to fix the address space layout at compile time. Micro-controllers usually do so because the RAM usually is embedded on-chip or on-circuit. For a modern computer, this approach is not feasible. The program would have to be compiled anew separately for every different computer setup. Therefore, the CPU provides mechanisms to deal with these issues. It offers the programs a virtual address space. The virtual address space is a range of virtual addresses, which map to the physical memory. This way, it allows a program to run independently of the physical address space organization.

Memory Segmentation [39, p. 5, pp. 65ff.] is one way to create an address space. The CPU splits the address space into segments. Segment registers reference the segments. Logical addresses with segmentation contain the base address

as well as information about the segment. Memory segmentation enables the use of address-layout-independent executables. With memory segmentation, an executable uses logical addresses to refer to memory. The operating system then maps the segment base addresses to linear address space chunks within physical memory. Whenever a program dereferences a logical address, the CPU adds the corresponding segment base address to the address. The linear address either points into the physical memory, or it points to virtual memory. A segment register either directly contains the address offset or point to a segment descriptor. The CPU supports various segment registers for the different program parts. Modern operating systems barely use segments. ARM does not support segmentation at all [57]. x86 provides registers for several segments [41, ch. 3.4.3]: Code Segment (CS), Data Segment (DS), Stack Segment (SS), Extra Segment (ES), General Purpose Segment F (FS), General Purpose Segment G (GS), and Task State Segment (TSS). In Real Mode, the segment registers contain the segment address offset. An x86_64 CPU also uses segment descriptors in protected mode and long mode. These descriptors contain the base address of the segment, its size, and some flags to customize the segment. Call gates are also realized via descriptors. Call gates allow the system to raise privileges by jumping to a call gate address. We detail privileges later in this section. Sets of descriptors are stored in descriptor tables. x86_64 supports a Local Descriptor Table (LDT), a Global Descriptor Table (GDT), and an IDT. The GDT contains code- and data segments that can be used by all tasks. It is located in the virtual memory, and the CPU looks it up via the Global Descriptor Table Register (GDTR). The LDT contains the per-process local descriptors. To get the correct descriptor table entry, a segment selector selects the index of the table. Figure 2.1 illustrates the linear address resolution of a logical address.

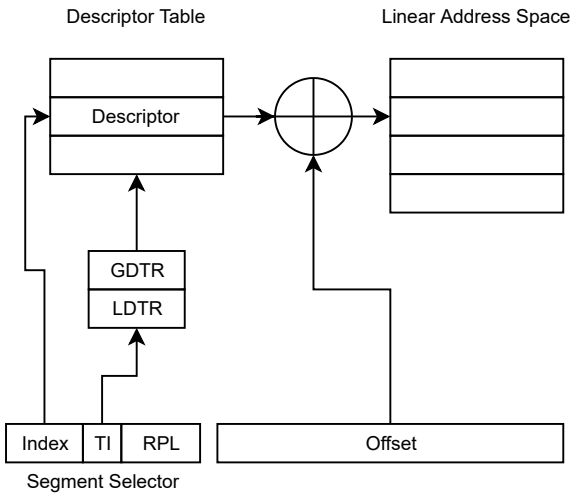

Figure 2.1: A x86_64 lookup on CPUs in protected mode with segmentation splits the logical address. The CPU looks up the target descriptor table by the Table Indicator (TI). The index selects the descriptor in the descriptor table. The segment base address of the descriptor is added to the offset to get the linear address [41, ch. 3.4].

In Long Mode, only the CS, the FS, and the GS and can be used [41, ch. 3.4.4]. The CPU ignores the base address, limit, and attributes of the ES, DS, and SS segments. The CPU treats them as if they have base address zero and max limit. Segmentation cannot be completely disabled on x86_64 in long mode.

Paging [39, pp. 7ff.] [41, ch. 4.1] is another memory management mechanism. Instead of using segments of variable size and offset, paging splits the entire memory into linear chunks of a fixed size. The chunks are called pages. The CPU then maps the physical pages to virtual pages in a tree-like structure. The virtual pages then form the virtual address space. A program in long mode or protected mode only operates with virtual addresses. The CPU is responsible for resolving virtual pages to physical pages. Therefore, it uses paging structure tables. Paging structure tables are an array of paging structure table entries. An entry contains the physical page number to the physical page or the next paging structure table and some flags. The CPU performs a virtual address resolution the following way: A control register is the entry point to the paging structure hierarchy. It contains the physical page number to the top-most paging-structure table. The CPU splits the virtual address into offsets. The top-most paging structure table is indexed by the top-most offset to get to the paging structure table of the next paging structure level. The lowest paging-structure level entry, usually named Page Table (PT) entry, contains the physical page number to the target physical page. The

rest of the virtual address is used as a byte offset within the page. The CPU uses flags within the paging structure entries to determine if the next paging structure table or the physical page is present. The paging structure can thereby create a sparse tree-structure to reduce the number of paging structure tables. x86 CPUs in protected mode and real mode and ARM [7, B4. 7] CPUs support paging. The x86_64 long mode has four page table levels [39, pp. 25ff.]: Page Map Level 4 (PML4) , Page Directory Pointer (PDPT), Page Directory (PD) and PT. These tables are 4 kB large, and each has 512 × 64-bit entries. Figure 2.2 illustrates a virtual address resolution on x86_64 long mode.

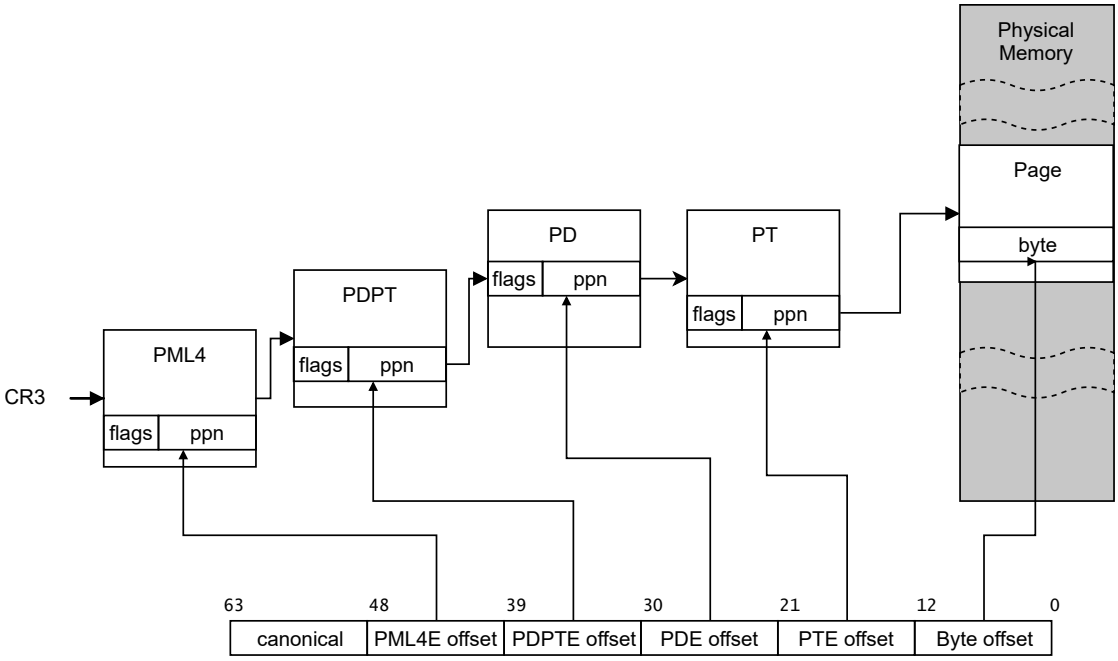

Figure 2.2: This figure shows the resolution of a virtual address on an x86_64 CPU in long mode. The CPU splits the virtual address into five parts to resolve the physical address: It uses bits 39 to 47 to index into the PML4 . The PML4 contains the Physical Page Number (PPN) to the corresponding PDPT. The CPU then uses bits 30 to 38 to resolve the PDPT entry, which contains the PD PPN. Bits 21 to 29 are used as the PD index, and the PT PPN can be looked up. Bits 12 to 20 are used for the PT index. The PT entry then contains the PPN for the actual page. Bits 0 to 11 are used as an offset within the page.

The address bits 48 to 63 are not used for the address resolution. x86_64 forces the upper bits, also called sign extension, have to be a copy of the bit 47. Such addresses are canonical addresses. If the system tries to access non-canonical

addresses, the CPU throws a general protection fault. As a result, the address space can be split into canonical lower addresses, non-canonical addresses, and canonical upper addresses, as shown in Figure 2.3.

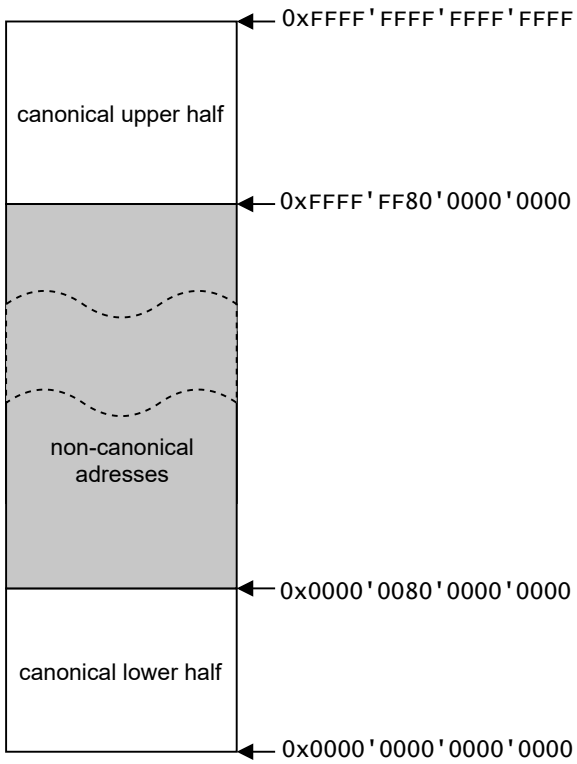

Figure 2.3: The canonical address space of the x86_64 long mode splits the virtual addresses into a lower canonical address space and an upper canonical address space. The non-canonical addresses in-between cannot be mapped.

Paging structure table entries contain flags in addition to the physical page number to the next mapping level. These flags provide additional information to the CPU or can be used to give the software information about accesses. Every mapping layer contains different flags, each represented by one bit. The flags which are important for this thesis are [41, ch. 4.5]:

- Present. The present-bit is zero if the page referred by this entry is not present. This way, not the entire virtual address space has to map to physical addresses.
- Read/write. If the writable-bit is zero, writing access to the mapped address space is not allowed. Read accesses are allowed in both modes.

- User/supervisor. If the user-accessible bit is not set, accesses of unprivileged programs to the corresponding part of the address space are not allowed.
- Accessed. The accessed bit is set by hardware or software and is reset by software. It indicates whether a read or write access has happened to a virtual address.
- Dirty. Similar to the accessed bit, the hardware sets the dirty bit when a program performs a write to the target address range of the paging structure entry.
- Page size. A PDPT entry or PD entry can directly refer to a huge page (often also referred to as a large page) instead of to the next mapping level. All lower bits of the address are then used as an offset within the physical memory. This way, the page table memory overhead is reduced for large chunks of physical memory that are linearly addressed. A huge PD entry refers to 2 MB of physical memory (12 + 9-bit address offset). A huge PDPT- entry with page size set addresses 1 GB (12 + 9 + 9-bit address offset) physical memory.
- No execute. If this bit is set, the mapped memory is not allowed to be executed [41, ch. 4.6].
- Global. The mapped pages are not thrown out of the caches when the system changes the virtual mapping space. The global flag indicates that a virtual memory area keeps maps to the same physical pages for different programs. The CPU does do not need to flush corresponding cache entries when it switches between different programs.

Whenever an address is resolved, the CPU checks if it maps to physical memory, and if the current executable is allowed to access this data by the given bits. It also updates the accessed and dirty bits. If the access can or shall not be performed, a fault is triggered to resolve the issue.

The CR3 contains the PPN of the PML4 in bits 12 to 50. When a program wants to switch the virtual address space, the CR3 is updated. All corresponding hardware buffers and caches are flushed implicitly by setting the CR3 . The page translation type is selected using the Control Register 4 (CR4) . Newer ARM versions provide two registers to refer to the top-level page table, namely Translation Table Base Register (TTBR) 0 and 1 [7, B4.7.1]. TTBR 0 usually contains the user mapping, and TTBR 1 the kernel- and IO mapping.

Interrupts and Exceptions are mechanisms provided by most CPUs and microprocessors. They allow the system to suspend the current program execution and

jump to a handler routine. The routine then handles the cause of the interrupt or exception and then continues the program. We distinguish three main types of x86_64 CPUs [39, ch. 8]: external interrupts, internal interrupts, and exceptions.

External interrupts are interrupts that are triggered by external IO or internal CPU features. They allow reacting to the change in peripheral states. Instead of hardware interrupts, the system can also poll for IO changes. Polling means periodically checking whether an event occurred. Polling is time-intensive since it blocks the CPU, and thus is avoided in most circumstances. A typical IO hardware interrupt would be a keyboard interrupt. On every key press and key release, an interrupt is triggered. Timer interrupts are CPU features. They interrupt the program flow after a specified time. Hardware interrupts are asynchronous interrupts. The interruption is independent of the program status. x86_64 CPUs handle external interrupts via the local Advanced Programmable Interrupt Controller (APIC), or via pins directly connected to the processor [41, ch. 6.3.1, ch. 10] [39, ch. 16]. The local APIC is integrated into the processor and is connected to the I/O APIC. When an external interrupt is triggered, the I/O APIC forwards it to the local APIC through the system bus or a specific APIC bus.

Exceptions are raised whenever an instruction causes an abnormal condition. An exception can happen due to invalid instructions, forbidden operations, or invalid memory or register accesses. Exceptions themselves are categorized in faults, traps, and aborts [39, ch. 8.1.3]. Some important exceptions in long mode are:

- General Protection Fault. This fault indicates a general access violation caused by a program, e.g., due to a lack of privileges or access to a non-canonical address.
- Page Faults. When the system tries to fetch data from pages that are not mapped or not allowed to be accessed this way (e.g., write to a read-only page that is marked as read-only), the CPU triggers a page fault.
- Double Fault. If an interrupt or exception produces an exception itself, the CPU triggers a double fault.
- Triple Fault. When the double fault causes an exception, a triple fault is triggered. At this stage, the system can most likely not recover any longer. So the triple fault restarts the system, instead of handling it via a software routine.

Exceptions are synchronous because they may directly interfere with the program execution.

Software interrupts allow programs to trigger interrupts by software. The x86 ISA has dedicated instructions to enter and exit interrupt routines from software. Interrupt routines run with higher privileges than regular programs. So they are often used as an interface between privileged- and unprivileged programs. x86_64 has its own set of instructions to enter and exit software interrupts. The IDT contains the interrupt and exception routine entries points as entry gates. Whenever the system triggers an interrupt or exception in long mode, the interrupts are disabled so that no other interrupt can intercept the routine. Information as the Register Instruction Pointer (RIP), CS, FLAGS Register E (EFLAGS), Register Stack Pointer (RSP), and SS are pushed onto the new stack. Then the target RSP is load from the TSS. The SS is then set to zero. The new entry is loaded into the RIP register, and the routine is then run. When the interrupt routine is finished, the same procedure is done in reverse. The x86_64 architecture introduces a new way of how to use stacks during interrupts, namely via the Interrupt Stack Table (IST). The IST is used to automatically load the corresponding stack for an interrupt or exception, instead of loading a stack from the TSS. x86 allows masking (disabling) and unmasking (enabling) of specific interrupts, *i.e.*, interrupts can be deactivated and re-activated. Not all interrupts are maskable (e.g., double faults and page faults cannot be masked).

Access privileges determine which registers or address ranges can be accessed by a program in which way. Not every program needs to have access to every system resource. If there is no limitation on access, malicious or bad programmed software could harm the system or other programs. They could also obtain secret information. Because of that, access to not required resources can be restricted by the hardware. The resources which have to be protected against forbidden access include:

- Registers. Especially control and status registers need to be protected.
- Memory. Not every memory region should be accessible to every program.
- IO. Not every program should be able to access IO addresses and registers.
- Instructions. Not every instruction should be allowed for every program.

For this purpose, x86 offers four different privilege modes, which are also called rings. Privilege level 3 offers the least privileges. It is also referred to as user mode. Privilege level 0 offers the full system features and is named supervisor mode. The privilege levels 1 and 2 offer some privileges and are usually not used on most operating systems. An exception is para-virtualized hypervisors. There, the hypervisor executes on privilege level 0, and the operating system runs on privilege level 1 or 2 [100]. The CPU manages privileges of resources by the

Descriptor Privilege Level (DPL) of gate descriptors, segment descriptors, and paging structure flags. Each core has an assigned Current Privilege Level (CPL). Each program can only access data, execute instructions, or trigger interrupts if the DPL of the target segment or gate descriptor is bigger or equal to the CPL. When a program successfully calls or triggers a call gate or interrupt gate, the CPL is raised to the DPL of the target CS.

Privileged Calls. Interrupt context switches and returns are rather slow. The x86_64 architecture provides the `SYSCALL`, `SYSRET`, `SYSENTER`, and `SYSEXIT` instructions [41, ch. 5.8] [39, pp. 448-459]. `SYSCALL` and `SYSENTER` allow a user program to call a privileged function fast, and `SYSRET` and `SYSEXIT` to fast return to user mode. The entry points to the privileged functions are set up via Model-Specific Registers (MSRS) [41, ch. 35].

`SYSENTER` and `SYSEXIT` are system call instructions to enter and exit a long-mode kernel from a 32-bit program run in compatibility mode. `SYSCALL` and `SYSRET` are the 64-bit fast system call entry and exit instructions. The following paragraph gives an example of `SYSCALL` and `SYSRET` on Intel CPUs [41, ch. 5.8.8]:

When a thread calls the `SYSCALL` instruction, the CPU stores the user thread's RFLAGS and the RIP (pointing to the next instruction) to registers. It raises the privilege level to 0 and sets up the CS, RIP, RFLAGS, and RSP, stored in the MSRS. `SYSRET` switches back to user mode. It restores the user thread CS and SS from MSRS, switches the privilege level to 3, and restores the RIP.

AMD and Intel have different compatibility system call schemes to switch between a protected mode 32-bit user program and a 64-bit long mode kernel, and different MSRS to support system calls. Intel x86_64 CPUs support `SYSENTER` and `SYSEXIT` instructions both in protected and long mode [41, ch. 5.8.7.1]. AMD, on the other side, does not support `SYSENTER` and `SYSEXIT` in long mode [39, ch. 6.1.2]. They instead have a 32-bit `SYSCALL` and `SYSRET` compatibility mode [39, ch. 6.1.1].

Caching and Buffers. Memories with a high density like RAM are cheap, but also very slow. It takes several cycles to fetch data from memory. Memory caching fixes this bottleneck. Caches are memory blocks which are faster than RAM, but also more expensive. Therefore, hardware caches are not directly used for data storage. Instead, they are used as a layer between the memory and the CPU. One memory cache entry can represent one memory data set, but various memory data-sets can map to the same cache entry. Caches use memory access patterns to predict which data will be used in the future and store it. The used memory

access pattern for caches is based on the principle of locality. Therefore, caches store the last data accessed because it is likely to be used again soon.

When the CPU then accesses memory, the cache is looked up before. If the required information is present within the cache, the cache lookup produces a cache hit. The CPU can load data directly from the cache and does not need to load it from the slower memory. If read access happens, and the data is not present any longer or has become invalid, the data has to be loaded from the RAM. When the data is available, it is used by the CPU and additionally stored into the cache. If there is already another entry present, it is overwritten. There are different policies if data is written: The write-through policy directly writes data into RAM and updates the cache entry. The write-back policy first writes the data into the cache only and updates the RAM upon cache eviction. Caches are indexed using virtual or physical memory addresses. In case the virtual address space is switched, virtually tagged caches have to be flushed.

Most x86 CPUs have different layers of caches. Bigger and slower caches are closer to the RAM, and small, but fast caches are closer to the core. Many x86 CPUs provide three layers of cache, namely L1, L2, and Last Level Cache (LLC) [44, pp. 2-23]. The L1 cache is split into instruction and data cache. L2 cache and LLC share instruction and data. The L1 and L2 cache are tightly coupled per core. The L3 is shared between all cores. The L1 and L2 cache on x86 systems is virtually indexed and physically tagged. Higher cache levels are usually physically indexed and physically tagged.

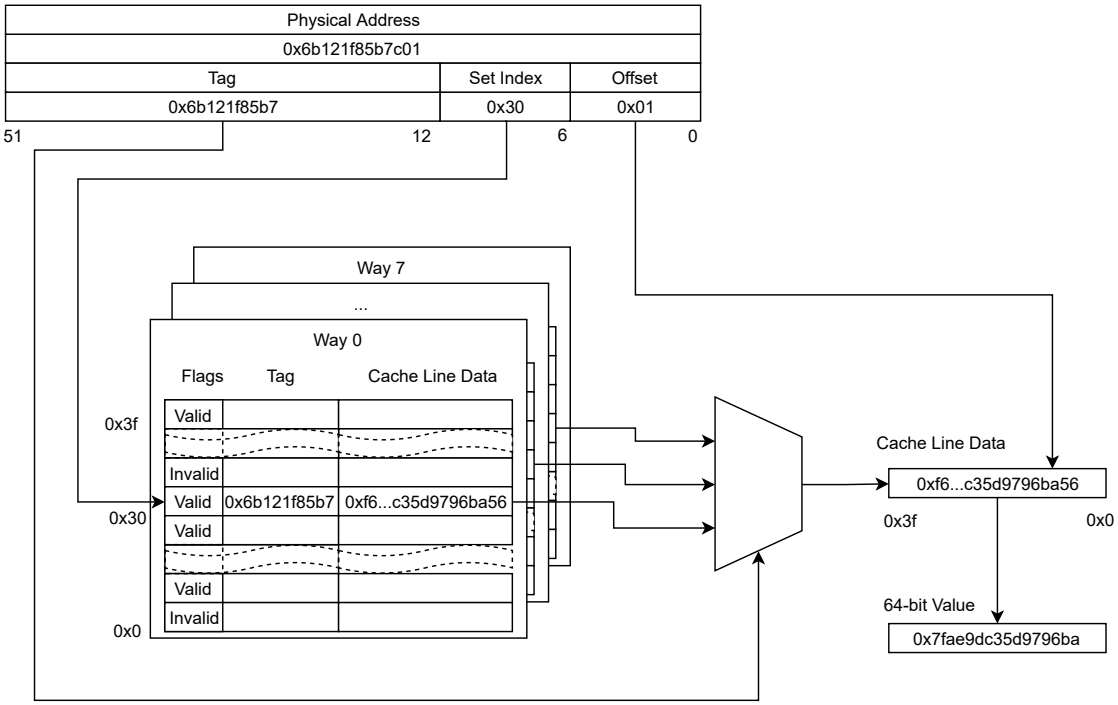

Figure 2.4: An example L1 cache lookup on an Intel Core i7 CPU [41, tab. 11-1]. The data cache is a 32 kB 8-way-set-associative cache with 64 B line width. The cache splits the physical address into the cache tag, set index, and line byte offset. The cache controller looks up the set via the index. It checks if any cache way in the set is marked as valid and if any valid cache way tag is matching. In this example, the cache way 0 entry is valid and matches. The cache controller looks up the 64-bit value stored at the corresponding byte offset in the matching cache line.

TLB. In addition to memory caches, CPUs provide TLBs to speed up the address translation. Some CPUs have split TLBs for instruction and data [44, pp. 22-23] [41, ch. 11.1]. Whenever the system has to resolve the mapping between virtual and physical addresses, the TLB performs a lookup. The entry consists of the virtual page number, the corresponding physical page number, and some mapping flags for access and privilege checks. If a mapping entry is present, the physical page number can be directly used to resolve the physical page, instead of walking through the four mapping levels in long mode. Most SMP CPU cores have dedicated TLBs for instructions and data. On Intel x86_64, the TLB can be tagged using the Process-Context Identifier (PCID) [41, ch. 4.10.1]. The PCID is represented by the least 12 bits of the current PML4 physical page number.

The entries of the TLB do not have to be flushed every a context switch happens. However, the entries may have to be removed if there is a tag collision for two or more PCIDs. Intel x86_64 CPUs also have their dedicated data and instruction TLBs for large pages [41, ch. 11.1, ch. 11.11.9].

Paging-Structure Caches extend the TLB to more layers to speed up the virtual address resolution further [41, ch. 4.10.3]. Instead of fully resolving to the target physical address, paging structure caches just resolve parts of the address translation by the corresponding virtual address. The paging-structure cache entry stores the physical page number of the next paging level as well as some flags to check for an access violation. The paging-structure caches are filled whenever a virtual address is resolved. The activity diagram in Figure 2.5 illustrates a TLB and paging-structure cache lookup of a virtual address. The PML4 cache takes the bits 39 to 47 of the virtual address as the index and returns the reference to the PDPT. The PDPT entry cache takes the bits 30 to 47 as the index and returns the PD physical page number. The PD entry cache is indexed by the bits 21 to 47 and provides the PT page number. These caches have to be flushed at every virtual memory switch unless they are tagged with the PCID [41, ch. 4.10.1].

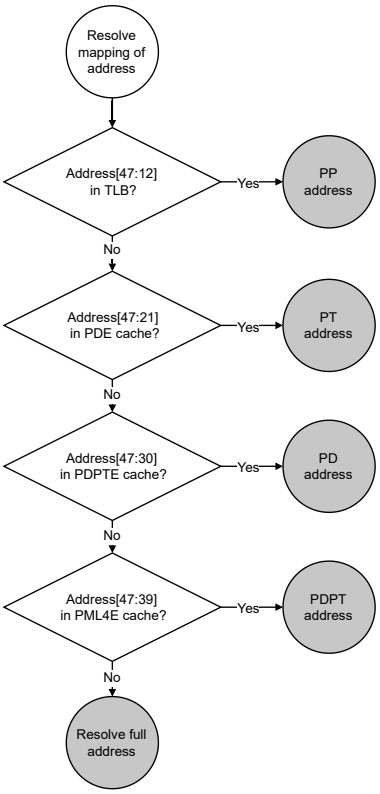

Figure 2.5: The address resolution of x86_64 in long mode with Paging-Structure Caches [41, ch. 4.10.3.2]. When the CPU resolves an address, it first looks up the TLB. If the TLB misses, the processor looks up the corresponding entries in the paging-structure caches, *i.e.*, the CPU looks up the address in the PT entry cache, the PD entry cache, PDPT entry cache, and PML4 entry cache. If one of them resolves the target paging structure, the result is taken as part of the page resolution.

Superscalar CPUs. Modern CPUs have a variety of measures to increase computational performance. A normal pipeline sequentially fetches instruction into the pipeline. A CPU can have a superscalar architecture to speed up the instruction pipeline [94, ch. 1.3.1]. A multiscalar CPU has multiple functional units that serve a special purpose (like an Integer Unit or a Memory Unit). The CPU fetches multiple instructions and distributes the execution of the different functional units [92].

Out-of-order execution is a superscalar processor mechanism that speeds up program execution. The CPU reorders and parallelizes sequential instructions

and executes them ahead of time to optimize the load of different CPU execution units. The CPU evaluates if an instruction can be pre-processed depending on the data flow graph. If data dependencies and currently available resources allow it, the CPU executes the instruction ahead of time.

Branch Prediction. When the program reaches a conditional branch, the CPU cannot out-of-order execute further instructions as the branch target is not known until the branch is processed [96]. In this case, the CPU can speculate what the branch target will be and execute instructions of this branch target to optimize the CPU load. The speculative results are temporarily stored. Once the CPU evaluates the conditional branch, it takes the results. Otherwise, the CPU discards the results. A Branch Prediction Unit (BPU) decides which branch shall be pre-executed. It estimates which branch will most likely be entered based on the previous branch targets [78]. The history of recent branch targets is stored in a Branch Target Buffer (BTB) [24]. The BTB entries contain the source address, target address, and a history of the last conditional or unconditional branches. This way, a branch can be estimated before it is happening, and the instruction pipeline of a processor can be filled earlier. If the branch prediction misses, the pipeline has to be flushed and reloaded. The BTB structure is similar to those of caches [24]. It uses virtual branch addresses as the index and provides the history of the recent branch targets from this point.

Intel Transactional Synchronization Extensions (TSX) [44, ch. 12] is a thread-synchronization mechanism introduced by Intel 2012 in the Haswell architecture. A thread on a multi-threaded system prevents race conditions on shared data by locking critical regions where other threads shall not access or change the same data. Locking is an expensive operation, which requires the use of a synchronization mechanism like semaphores or mutexes [94, ch. 2.3]. Intel TSX is an instruction set extension that speeds up atomic data access. A software developer can define atomic code regions. The system tracks the memory accesses within these regions. If another thread accesses the memory at the same time as the current thread and produces a read-write conflict, results are discarded. This way, operations can be performed atomically without locking the shared resources. Unlike other mechanisms, a failing transaction inside an Intel TSX region does not trigger an exception that can be caught. Instead, the event is handled in the TSX region itself. TSX includes two features: the Hardware Lock Elision (HLE) instruction extension for backward-compatible binaries, and the Restricted Transactional Memory (RTM) instructions. HLE adds two new instruction prefixes `XACQUIRE` and `XRELEASE`. With HLE, a locking mechanism can be implemented. On new systems, the HLE lock marks the start and stops a TSX region. At the first entry

of the region, the optimistic program skips writing to the lock variable. If the transaction fails, the region is restarted again, and the lock is acquired. The HLE extension is ignored on CPUs that do not support TSX. So binaries with `XACQUIRE` and `XRELEASE` can have locks that run on CPUs with and without TSX. The RTM instructions `XBEGIN`, `XEND`, and `XABORT` allows a developer to implement a TSX region with an additional fallback option. An atomic operation region is started at `XBEGIN` and ends with `XEND`. If the memory transaction fails in an atomic region, the CPU reverts the state. It then returns to `XBEGIN` and writes an error code to `EAX`. The program can then handle the failed transaction.

Prefetch. The x86_64 `PREFETCHh` instruction allows a program to fetch data into the caches before it is accessed. Developers can use this feature to pre-load data in order to boost the actual access of the data when it is needed. The data can be loaded into different cache levels. Because a user program could try to prefetch not-yet mapped data, the `PREFETCHh` instruction may not fail. A prefetch attempt of data that is not accessible or requires higher privileges does not raise a page fault or general protection fault.

2.2 The Linux Operating System Kernel

A kernel is the core part of an operating system. It is capable of booting, providing a generic interface to the system, managing the user programs, protecting the systems from corruption and attacks, and many other features. This section gives an overview of the Linux kernel and its parts that are relevant for this thesis.

The Linux kernel. In 1991, Linus Torvald released the first version of the Linux kernel for the Intel 80386 CPU [50, p. 6]. The Linux kernel is open source. Other operating systems share common principles with the Linux kernel but are often held close source. The Linux kernel is a monolithic kernel [89, p. 145]. A monolithic kernel, in comparison to a micro-kernel, runs its functionality in a single program. Micro-kernels, on the other hand, break their functionality down into separated processes. The Linux kernel uses a Portable Operating System Interface (POSIX) compliant interface [89, p. iv]. It can be cross-compiled for different system architectures, like x86 and arm [67, p. 3]. This master thesis focuses on the Linux kernel version 4.10-rc6 on x86_64 with default kernel configurations.

Kernel Boot. The operating system boot consists of multiple steps. x86 CPUs support one of two main processes to load the operating system into RAM upon

startup: The legacy Basic Input/Output System (BIOS) boot process and the newer Unified Extensible Firmware Interface (UEFI) boot process. The UEFI or BIOS firmware loads a boot loader like the GNU Grand Unified Bootloader (GRUB) from the drive or network in multiple stages [19] [48]. The last boot loader has a driver for a file system. It loads the Linux kernel image and the initial ramdisk into RAM and executes the kernel [62]. The Linux kernel can be packed in different file formats. A common kernel image file format on x86 platforms is bzImage [91]. A bzImage consists of a header, setup code for initial execution, and the compressed `vmlinux` image. The boot loader starts the Linux kernel head code in Real Mode. After the basic setup, the kernel switches to either Protected Mode or Long Mode. Then, the kernel decompresses the `vmlinux` binary and executes it. The start_kernel function [63] is the platform-independent entry point to the uncompressed kernel. This routine initializes the entire kernel infrastructure and drivers, the interrupts, and the scheduler. Then, it starts the kernel `init_thread`, the kernel worker thread daemon (`kthreadd`), and the idle thread. The kernel `init_thread` sets up SMP and then executes the first user process.

Processes and Threads. Linux handles its user programs as processes. A process can have one or more threads. The threads of a process share the same virtual memory, but all have their stack and registers. A process forks itself to create a new child process. The forked process has the memory layout, files, and the same threads as its parent. The created child process can then execute another binary using `execve` [50, pp. 31-32]. The Linux kernel abstracts and manages all resources for processes and threads. It provides drivers and unified interfaces to the user processes and threads. A user thread can access these resources via system calls or memory mapping. In addition to user threads, the Linux kernel also supports kernel threads. The kernel thread daemon creates the kernel threads.

Each thread has its own kernel stack and a `task_struct`. The `task_struct` contains all relevant information for the task. The Linux kernel handles thread execution via a scheduler. The scheduler is responsible for switching between threads, balancing the processing time of processes, and optimizing the CPU load [64]. The x86_64 Linux kernel supports program architectural backward compatibility for processes compiled for x86_32.

Memory Layout and Management. The Linux kernel v4.16-rc6 for x86_64 long mode uses different names for the paging structures as Intel or AMD - The PML4 is the PGD, the PDPT is the Page Upper Directory (PUD), the PD is the Page Middle Directory (PMD). A PT is still called PT. In the context of Linux, we will use the Linux paging structure definitions. The Linux kernel uses the upper

canonical address space as kernel space and the lower half as userspace. Figure 2.6 gives an overview of the kernel virtual memory layout.

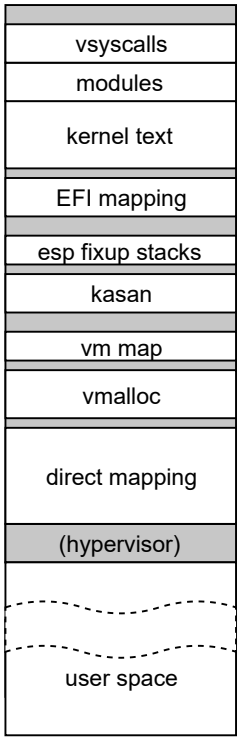

Figure 2.6: The Linux kernel v6.10-rc6 virtual memory layout [51].

The upper canonical half of the virtual address space is reserved for the kernel. It starts with a guard hole that can be used for a hypervisor. The direct mapping is an identity mapping of the physical address space for direct virtual access. The `vmalloc` area is the kernel heap memory, as well as the IO remap area. The virtual memory map area contains a linked list of the physical pages and their virtual addresses. The KernelAddressSanitizer (KASAN) region contains the shadow memory table for dynamic memory access error detection. The `ESP` fixup stacks are workaround stacks to not leak parts of the `ESP` register content to 16-bit mode user programs [5]. The Extensible Firmware Interface (EFI) mapping space is reserved to map EFI runtime services for EFI runtime calls. The kernel text section contains the Linux kernel RO binary. It is mapped starting from physical page 0 [51]. The modules are mapped next, followed by the virtual syscall area. The kernel maps data and functions into the virtual memory and makes them accessible to user threads. User programs can directly access kernel data (e.g., date and time information) or call functions without switching to kernel mode [11]. The virtual syscall mechanism is deprecated and replaced by the vDSO mechanism.

The kernel still reserves the virtual memory region to virtualize the virtual system calls for backward compatibility with old user programs. The initial PGD of the Linux kernel is set up at boot and used as the basis for all paging structures. The kernel copies the content of the upper PGD half when a process is forked. The kernel also keeps the kernel virtual memory in sync, so all processes share the same virtual kernel memory.

The physical memory is managed using a physical memory map with `struct page` entries [88]. Linux supports different memory models to maintain the memory map. On x86_64, the Linux kernel uses SPARSEMEM memory management [86]. The memory is represented in sections of different sizes, which then resolves to `page struct` entries. Each Page-Frame Number (PFN) refers to a physical page. Its upper bits are used to find the section, and the lower bits find the corresponding page. The Linux kernel linearly maps the entire physical memory map into the virtual memory to speed up the translation between PFN and `page struct`. Holes between segments are not mapped in the virtual memory map, but their virtual memory within the map is still reserved. The kernel uses the virtual memory map offset to convert between a PFN and its `page struct`.

The virtual memory information of each Linux process is held in a `struct mm_struct`. It contains the reference to the PGD and a double-chained sorted list of the virtual memory areas. These virtual memory areas each represent a linear virtual memory area within the address space. They contain the reference to the source where they are loaded from (e.g., a file), as well as access flags. A page not representing a file is an anonymous page, and pages that contain file contents are page-cache pages. The kernel loads virtual memory area pages on demand. When a virtual memory area is created, the kernel just refers to the corresponding file but does not allocate and map physical memory yet. When a thread tries to access the page, a page fault is triggered, and the kernel maps the page on-demand. Besides, the kernel shares pages where possible. As long as memory content is not modified, it can be mapped into multiple virtual memory areas simultaneously. An example is the virtual memory of a forked process. When the parent process forks its child, the physical pages are not duplicated. Instead, the kernel links them into the new page table structure and sets the page table entries non-writable in both processes. When a thread attempts to write to a deduplicated virtual memory area, the CPU triggers a page fault, and the kernel performs a Copy-On-Write (COW). The kernel copies the physical page content onto a new page, links the new page, and sets the new page table entry writable again. The old page table is also made writable again if it is the last one referencing the deduplicated page. Pages can even be deduplicated at runtime via Kernel Samepage Merging (KSM) [18]. Kernel Space scans over anonymous user pages and deduplicates them if possible.

The scheduler only switches the virtual memory layout only if necessary. If possible, it does not replace the current paging structure to avoid unnecessary cache flushes and reduce the context switch overhead [97]. If the scheduler switches to a new thread that does not need to access the user virtual address space, the scheduler stays with the current virtual address layout. The thread can keep the foreign virtual address layout. Since all threads share the kernel mapping, the kernel-only threads can always stay with the current virtual memory layout.

Per-CPU data. The Linux kernel supports SMP CPUs. Therefore, it requires to manage per-core instantiated data, the so-called per-CPU data [2] [56]. An example of per-CPU data is the IRQ stacks or the GDT. These data structures are virtually mapped into the kernel space and set up at boot. The Linux kernel provides functions to read/modify/write per-CPU data of the current core, another core, or all cores. Linux uses the GS to select the per-CPU defined variables per core in kernel mode. In user mode, the GS is zeroed.

Interrupt, System Call, and Exception handling. Whenever an x86_64 interrupt or exception is triggered, the CPU looks up the corresponding IDT entry and checks the DPL. If the interrupt descriptor is valid, the CPU calls the entry gate by the segment selector and the offset. All interrupts, system calls, and exceptions which direct to the Linux kernel code segment have their entry and exit points in `arch/x86/entry/entry_64.S` and `arch/x86/entry/entry_64_compat.S` [59]. These entry and exit points are stored in a separate Linux kernel text section. The Linux kernel has to provide different types of entries due to x86_64's interrupt, system call, and exception calling conventions [9].

System Calls. A Linux system call, or syscall, is a mechanism that allows a user program to access kernel functionality in a defined way. User programs usually call system calls via a wrapper library like `libc` [15] [68]. Libc provides a POSIX-compatible interface to the user program for independence and prepares the system call in user mode if required. Performing a system call into the kernel is expensive. System call functionality that does not need additional privileges can be entirely handled in `libc`, `vsyscall`, or `vDSO` [11]. The user program calls the `vsyscall` or `vSDO` without a need for a kernel gate or kernel system call, which is much faster. The kernel maps the `vsyscall` segment into the top of the virtual kernel space and makes it user-accessible. The `vDSO` segment, on the other hand, is dynamically mapped into the userspace. If the system call requires increased privileges, the user program has to enter the kernel.

A user thread passes a system call number and up to six arguments to the kernel via registers when it performs a system call. The system call number is determining which actions shall be performed. After storing the necessary information into the registers, the user thread enters the kernel by executing the `SYSCALL` instruction, the `SYSENTER` instruction, or the software interrupt `INT80`. Since these ways of entering the kernel all need to be handled differently, the Linux kernel has a different entry point for them. The long mode `SYSCALL` instruction calls `entry_SYSCALL_64`. On AMD CPUs, the `SYSCALL` instruction is also used by 32-bit protected mode user programs. They enter `entry_SYSCALL_compat`. Figure 2.7 shows the flow chart of `entry_SYSCALL_64`. 32-bit programs that are running on an Intel x86_64 CPU enter the Linux kernel in `entry_SYSENTER_compat` via the `SYSENTER` instruction. Legacy protected mode programs perform a system call via `INT80`, and land in `entry_INT80_compat`. To support protected mode programs on both Intel and AMD CPUs, Linux recommends entering the kernel from a protected mode thread via `__kernel_vsyscall`, which is mapped by `vDSO` [75].

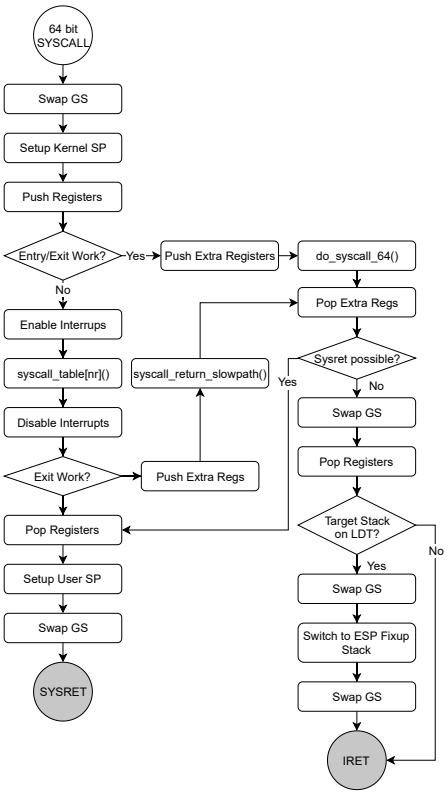

Figure 2.7: The flow chart of the Linux kernel v4.10-rc6 64 bit system call entry (without tracing) [59][66]. The long mode `SYSCALL` entry provides a fast entry path and a slow entry path. The kernel entry code switches to the kernel GS. It sets up the kernel stack pointer and pushes the necessary registers on the stack. The registers `R12 - R15`, `RBP`, and `RBX` are preserved by the calling user thread and do not need to be saved [66]. If the current thread requires to perform syscall entry or exit work, it enters the slow path, stores the rest of the registers, and calls `do_syscall_64`. If no entry or exit work is pending, the thread directly calls the system call function from the system call callback table. The thread can return to user-mode in two ways. If possible, it returns via `SYSRET`, because it is much faster. If a return via `SYSRET` is not possible (e.g., the target return address does not match the entry address stored in `RCX`), the thread has to exit the system call via `IRET`. The kernel switches the stack to the `ESP` fixup stack if the user stack segment is in the LDT before it executes the `IRET` instruction, and returns into user mode [5].

Interrupts and Exception Entry and Exit. The Linux kernel supports different kinds of interrupts and exceptions. On x86_64, interrupt gates are registered in the IDT. The Linux x86_64 interrupt layout is shown in Table 2.1. As described

Range	Interrupts
`0x0-0x1F`	System Traps and Exceptions
`0x20-0x7F`	Device Interrupts
`0x80`	Legacy `INT80` syscall
`0x81-0xEE`	Device Interrupts
`0xEF-0xFF`	Special Interrupts

Table 2.1: This table gives an overview of the x86_64 Linux kernel IDT layout [60]. The first 32 entries of the Linux x86_64 kernel IDT are reserved for system interrupts and exceptions, like the external Non-Maskable Interrupt (NMI) or the page fault. They are defined by the CPU [41, tab. 6.1]. The following IDT entries are device interrupts and exceptions, which are configured by the APIC. They are platform-dependent (e.g., driver interrupts). Interrupt `0x80` is reserved for the legacy `INT80` system call. Linux defines the last interrupts (`0xEE-0xFF`) as special interrupts, which are fixed by the kernel. An example for a special interrupt is the reschedule interrupt.

above, the x86_64 kernel has different entry points for the interrupts and exceptions. The use of the `SWAPGS` instruction is especially tricky. When an interrupt is triggered, the kernel has to determine whether the thread came from user mode or kernel mode. Linux has a fast check and a slow check mechanism to determine if the thread came from user mode or kernel mode [59]. In the fast check, the kernel reads the CS of the called thread and checks the DPL. The kernel performs the fast check on every entry that cannot interrupt in a context switch. The slow or paranoid entry check, on the other hand, is performed on interrupts and exceptions that can interrupt an ongoing entry handler. Exceptions and interrupts like NMI or INT3 (the breakpoint interrupt) may interrupt right between a CS check and SWAPGS of an already ongoing entry function. If the fast entry is performed in this case, the entry function switches the GS back to the user GS, and thereby accesses wrong per-CPU data. The entry code instead reads the GS from the MSRS to determine the thread status.

Device interrupts enter the kernel via an entry stub [66]. Each device interrupt has its dedicated stub. The stub pushes the interrupt number on the stack and jumps to a shared device interrupt function. `common_interrupt` saves the registers, and swaps GS if required (performing a fast GS check). It switches to the interrupt stack if the thread came from user mode and then jumps into the

`do_IRQ()` function. `do_IRQ()` then executes the actual interrupt routine based on the interrupt number. At the exit point, `common_interrupt` tests if the thread returns to user mode via a fast check. If it returns to user mode, it swaps back GS and returns to user context via an interrupt return instruction. If the thread returns to kernel mode, the current thread returns to the previous context or switches to another thread if the current thread terminates. The kernel handles special interrupt entry and exits similar to the device interrupts. Instead of entering the kernel interrupt routine via `do_IRQ()`, these interrupt entry functions directly call the kernel function.

System traps and exceptions have either a non-paranoid or a paranoid entry point. If a non-paranoid system exception handler is triggered, the CPU pushes all registers on the stack. If a fast check indicates the thread came from user mode, the kernel swaps the GS and calls the actual Interrupt Service Routine (ISR). If the thread came from kernel mode, the entry function performs additional fixes if it returns to kernel mode with the user GS set. That can happen if the interrupted thread is returning to user mode via an interrupt return instruction, or if it executes `native_load_gs_index` to update the GS of the current thread (e.g., to clean up the `mm_struct` of a terminating process).

A paranoid entry is used for all exceptions and traps that can interrupt at any time, so also at other atomic interrupt entry and exit points. When a thread enters a paranoid exception, it first performs a fast user mode check. If the thread entered from user mode, it switches the stack and performs a similar sequence as the non-paranoid exception entry. If the fast check indicates that the thread may be in kernel mode, the thread performs the paranoid check, and swaps GS is required.

The Linux kernel implements a particular entry and exit point for NMIs. It allows nesting NMIs. If the current thread comes from user mode, it switches its current stack to the thread stack. If the thread came from kernel mode, the interrupt entry routine sets up an NMI nesting stack frame that following nested NMIs can use to handle nesting.

Kernel Stacks. The Linux kernel supports different stacks for the kernel [1]. Each kernel thread and user thread has its dedicated 8 kB thread stack. That stack is used for thread work and to handle system calls. To keep the per-thread stack size small, Linux has additional individual per-CPU stacks. For device Interrupt Requests (IRQs) and software IRQs from the kernel, each CPU core has an interrupt stack with 16 kB size. When the thread came from user mode, the kernel switches the IRQ stack. The x86_64 architecture allows the kernel to automatically

load one of up to 7 stacks per core for interrupt descriptors to circumvent nested interrupt stack race conditions. These stacks are registered in the IST of the TSS and indexed via the interrupt gate descriptor. The Linux kernel uses the IST for special interrupts and exceptions. The kernel has a separate stack for double fault exceptions, NMIs, Machine Check Exception (MCE), and debug interrupts.

2.3 Kernel Memory Protection Principles and Recent Work

Operating system kernels nowadays provide a rich set of mechanisms to protect themselves against forbidden access and misuse. Monolithic kernels become more complicated with increased size. Therefore, developers cannot ensure to fix every weakness of the kernel and its modules. Kernel countermeasures make use of hardware features to prevent attacks on the kernel from a malicious user program.

Address Space Isolation. The kernel memory is split up into kernel space and user space to prevent programs from unallowed access or execution of kernel memory and resources. The Linux kernel is mapped into the upper half of the canonical address space on x86_64, and user programs have the lower canonical space available. Even though every process has its own virtual address space layout, they all share the same kernel address space. When a user program wants to access restricted resources, it has to do so via a defined entry point. Such an entry point can be e.g., a system call or an exception. The kernel then handles the request and passes it back to the user program. If a user program tries to access kernel memory, the CPU informs the kernel via a page fault [41, ch. 5.11.3, ch. 4.7]. The Linux kernel then informs the user program about its termination via a `SIGSEGV` (segmentation violation) signal [50, p. 393]. This signal kills the process if the process does not catch it via a signal handler.

SMEP. An attacker might use kernel bugs to manipulate return addresses on the stack or function pointers to execute malicious user code in kernel mode. SMEP is a security feature that is available on x86_64 CPUs. SMEP restricts the execution of non-privileged instructions in user mode. When SMEP is active, and the CPL is not in ring 3, executing an instruction on a non-privileged page will result in an exception. It is activated in the CR4 of x86_64 CPUs.

SMAP has been developed for Intel x86_64 CPUs, and is similar to SMEP. When the system is in supervisor mode, it is not allowed to access the virtual

pages marked as user pages. This feature can be activated in the CR4 and can be temporarily disabled by setting the `EFLAGS.AC` flag. If a program has to read or write user data, it can temporarily disable SMEP.

Write XOR Execute (W⊕X) . Since a user program cannot access the kernel memory directly, attackers try to find vulnerabilities in the kernel to inject code into the kernel space and execute it. An attacker can, for instance, produce a buffer overflow to overwrite functions, or write malicious instructions into the buffer and modify the stack return address. The W⊕X policy is a kernel mechanism that prevents such attacks. The x86_64 page structures support flags to make a page non-executable (XD) and writeable (R/W). The W⊕X policy defines that every page shall either be writable or executable, but never both. By restricting writable executable memory accesses, attackers can no longer inject malicious program snippets into the kernel and then execute them. There may still be memory regions that are neither writable nor executable. An attacker can thereby not inject code into executable regions any longer, and also not make the CPU execute buffer content.

KASLR. Although the previously introduced countermeasures hamper many attacks, attackers are still able to obtain information out of the kernel or even execute instructions in kernel mode [25] [87]. Return-oriented Programming (ROP) exploits like `return-to-libc` attacks usually require two pre-conditions: First, they need a weakness in the system under attack. Such weaknesses can be an unchecked boundary indexing in a buffer. The second requirement is the knowledge about the virtual address space layout of the system under attack. Address Space Layout Randomization (ASLR) is a method to mitigate attacks that need to have pre-knowledge about the virtual address space layout [76, 83]. When a process gets initialized or requests more memory, the kernel places the memory regions at random positions in the virtual memory address space. Since the attacks mentioned above need to know the virtual address space layout, ASLR impedes these attacks. KASLR uses the same mechanism as ASLR and randomizes the kernel virtual address layout upon boot. It was merged into the Linux kernel 3.14 [12]. Figure 2.8 explains how the virtual address space layout of an x86_64 virtual kernel memory is randomized. The boot loader places the kernel code at a 2 MB-aligned random address within the 1 GB kernel text region [61]. The kernel places modules with a random page-aligned base address between 4 kB and 2 MB [35]. The kernel relocates the direct physical mapping, the `vmalloc` region, and the virtual memory map during boot. The virtual address space in which they are placed is ≈119 TB large, and their virtual start addresses are aligned to

1 TB. To sum it up, Linux 4.10-rc6 KASLR for x86_64 adds ≈9 bit entropy to the kernel text virtual address, ≈10 bit entropy to the virtual address offsets of modules. The offset relocation of the physical mapping region, the `vmalloc` and I/O remap region, and the virtual memory map region add ≈7 bit of entropy each.

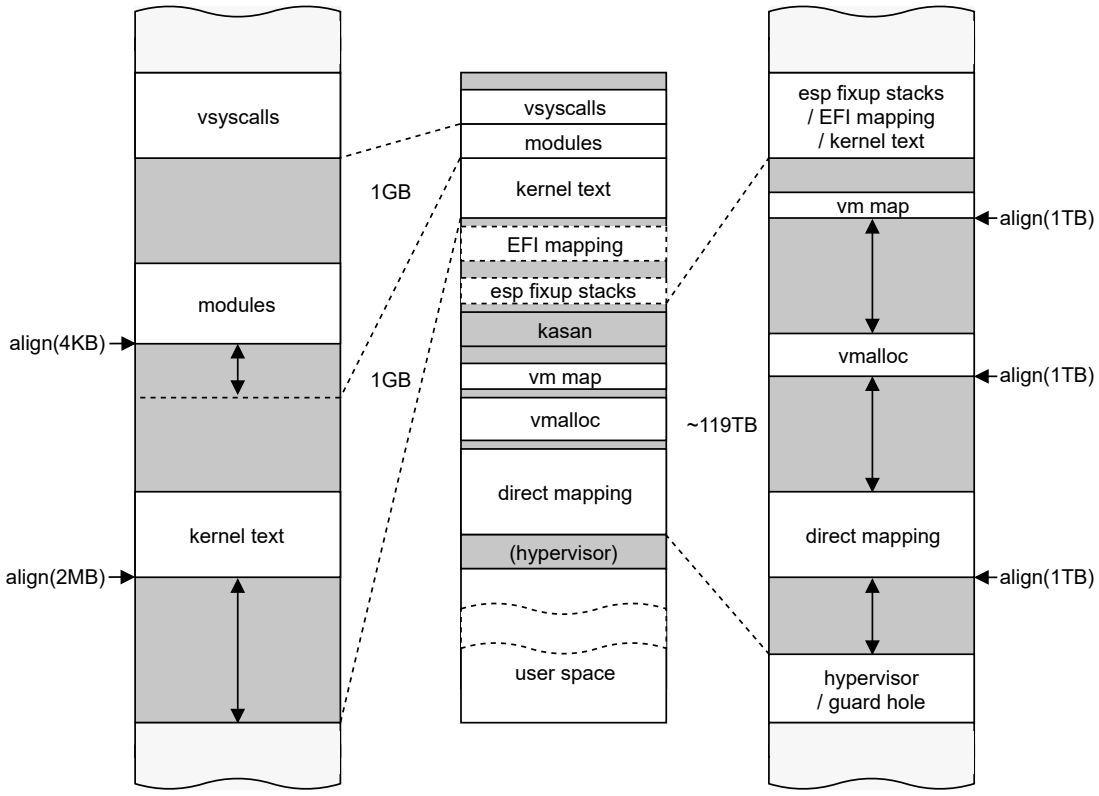

Figure 2.8: The KASLR of the Linux kernel 4.10-rc6 for x86_64 randomizes the offsets of the kernel text section, the module base address, the direct physical mapping, the `vmalloc` and `ioremap` region, and the virtual memory map memory at boot [51]. The KASLR region starts at the hypervisor or guard hole end address. The end of the KASLR region is the lowest start address of the kernel text section, the EFI mapping, and the `ESP` fixup stack (`ESP` fixup stacks and the EFI mapping are compile-time optional). KASLR and KASAN are mutually exclusive. Therefore, KASLR uses the KASAN shadow memory region to increase the virtual memory space and therefore the entropy.

2.4 Side-Channel Attacks

Kocher [53] introduced side-channel attacks [34]. A side-channel leaks information from a hidden state or sequence. A side-channel attack makes use of such a leak. The attacker analyzes the side-channel information to recover the hidden state or sequence of a victim. Such side-channel information can be a power consumption analysis, heat analysis, or timing differences. Spreitzer et al. [93] categorize side-channel attacks by the following parameters: The activeness of a side-channel attack shows how much the attacker influences the device under attack to leak information. The more passive an attack is, the harder it is to attack. An attack's invasiveness shows how much access or control the attacker requires to have on the target system. Highly invasive attacks even need to open the chip of the target under attack, while less or non-invasive attacks may not even need hardware access. Covert channels allow attackers to exchange information between two parties that are not allowed to communicate. Many side-channel attack techniques can directly be used to open a covert channel.

This thesis focuses on local software-based timing side channels on x86_64 machines that do not need direct hardware access. Such side-channel attacks are performed from a malicious process that is executed on the target system. They make use of the `RDTSC` instruction to read out the time-stamp counter and measure time.

Cache Timing Attacks. Cache attacks are a widely known technique to obtain information about data accesses of other processes or the kernel. Caches are usually smaller than the virtual or the physical address space. Therefore, caches use indexes and tags to map from an address to a data set. Since the cache entry index is usually smaller than the virtual address, only some bits of the virtual address is taken as the index, and the rest is stored in the tag. An attacker, who knows parts or the entire virtual address space layout, can provoke cache misses and measure the timing of memory access to estimate which data has been accessed. Since only some bits of an address are used to index caches, the attack on kernel data can be performed via congruent user addresses. Thus, the attacker only requires a process without privileges. Different cache levels can be attacked this way. In the following, we introduce some types of attacks which can obtain information about the kernel memory and its layout. These attack types use cache timing side channels.

Prime+Probe. Osvik et al. [82], Gruss et al. [27], and Younis et al. [102] use Prime+Probe for their attacks. A Prime+Probe attack targets data caches and allows them to check if another thread has accessed data. Prime+Probe first accesses all lines of a cache set. The cache lines within the cache set are filled with data of the attacker. The attacker waits until the attacked thread is executed and reaccesses the data. If the target has accessed or executed an address with the same cache set in-between, a cache miss happens because the cache line tag does not match any longer. The CPU has to reload the data from RAM. The attacker's data access takes longer, and the attacker learns if the victim accessed or executed the target address. An example cache set attack is illustrated in Figure 2.9.

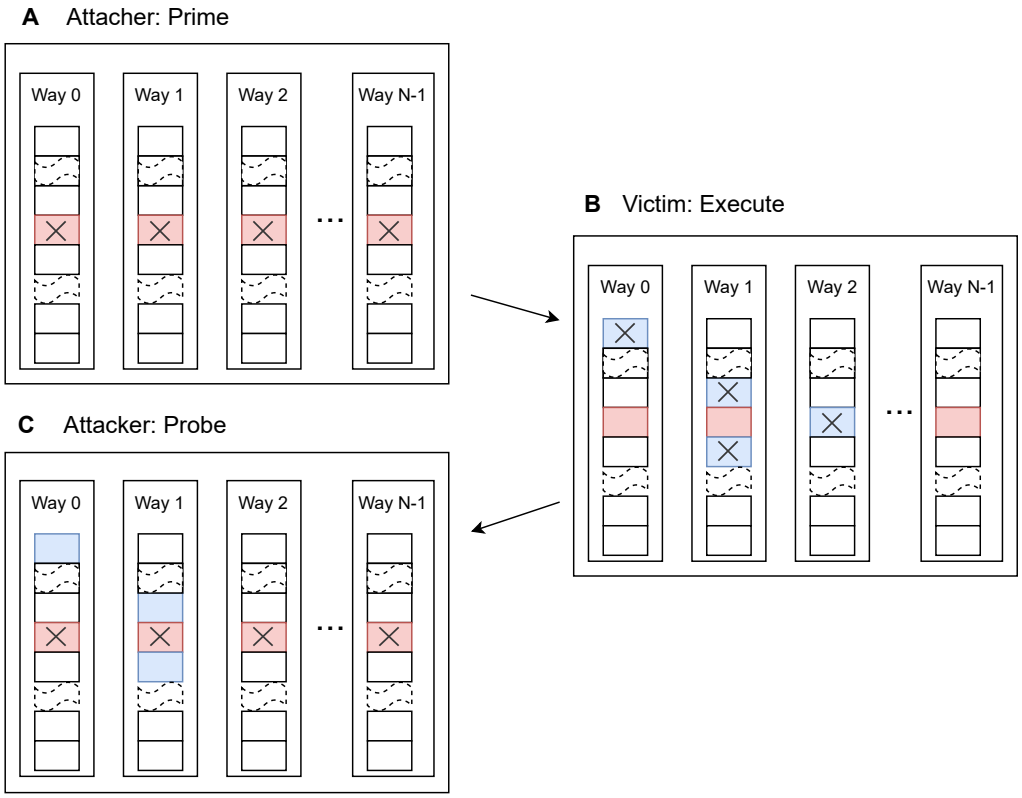

Figure 2.9: An Prime+Probe attack on a single cache set [34, p. 104]. Red cache line entries are loaded by the attacker, blue by the victim. Starting with the prime step (**A**), the attacker accesses indices of a data array that share the same cache set index. The CPU fills all cache lines of the set with the data from the array of the attacker. Then, the attacker lets the victim execute. When the victim tries to access or execute a physical address with the same cache set (**B**), the CPU gets a cache miss and loads the new entry into the cache. The attacker then performs the probe step (**C**). The attacker accesses all the data of the same cache set again and measures the timing. If one or more accesses take longer than a defined threshold, the attacker knows that the victim has accessed or executed lines from the same cache set. The attacker can thereby derive parts of the physical and virtual address of a data access or branch path of the victim.

Prime+Probe attacks can be performed on any level of data and instruction caches [74]. Prime+Probe can also attack cross-core on an SMP CPU if it attacks the LLC.

Flush+Reload. Another cache timing attack strategy is Flush+Reload. This attack is similar to Prime+Probe, but instead of accessing data in the first place, the cache lines under attack are reset using a flush instruction. This attack type was described by Yarom and Falkner [101]. A Flush+Reload works on the LLC cache lines for shared pages. Such a shared page may contain, e.g., a shared library. First, the attacker flushes a shared address. The CPU flushes the corresponding LLC cache line. Then, the attacker waits for the victim to access or execute the address under attack. Then, the attacker reaccesses the address. If the access time is below a pre-defined threshold, the CPU has loaded the content into the LLC in the meantime. The attacker can thereby find out if the victim has accessed the shared memory.

The **Flush+Flush** attack [29] is a side-channel attack that is similar to the Flush+Reload attack. Instead of reloading the address after the flush, the address is flushed again. The second address flush is faster when the data has not been cached in the LLC. The attacker can observe data accesses this way.

Prefetch Side-Channel Attacks. Gruss et al. [28] wanted to use the property of the prefetch instruction of Intel x86_64 CPUs to obtain information of the Kernel Virtual Memory (KVM) layout and the access pattern of kernel addresses. The prefetch instruction, as described in Section 2.1, does not fail. It even does not fault if the fetched virtual memory address does not map to a physical page or requires a higher privilege mode to access it. Gruss et al. [28] states that it instead just loads the data into the caches, even if the callee lacks permissions. Gruss et al. [28] presented two attacks that use `PREFETCHh` instructions to obtain information about the kernel virtual memory mapping from user mode. The CPU needs to resolve the physical address. It looks up the TLBs, the paging structure caches, and in the worst case, the paging structure tables to convert the virtual address to a physical address. The execution time of a prefetch instruction depends on the number of cache, buffer, and table lookups. The translation-level recovery attack measures this time to obtain information about the kernel virtual memory mapping. With that information, an attacker is able to recover the kernel virtual memory layout and break KASLR. Gruss et al. [28] also assumed the prefetch side channel can also be used to find the physical page of a virtual page in the kernel. The address-translation attack prefetches a virtual memory address. Then, it prefetches a target virtual memory address within the direct physical mapping. The prefetch instruction loads the value into the data caches at the first prefetch. If the physical address of the direct physical mapping matches the target virtual memory address, the CPU does not need to load the data caches again. Therefore the access is faster if the target virtual memory address refers to the physical address of the direct

physical mapping. The attacker is able to find the physical memory address of any virtual memory address in the kernel space or user space [28, p. 7].

Double Page-Fault Attacks. Hund et al. [37] found another side-channel attack on the kernel virtual memory layout. Their attack makes use of a specific behavior of Intel CPU TLB lookups. When a user program accesses privileged memory, the CPU triggers a page fault. On Intel CPUs (unlike, e.g., AMD [37, p. 9]), the CPU creates a TLB entry at a TLB miss even if the privilege check fails. The double page-fault attack, as shown in Figure 2.10, makes use of this mechanism to recover the virtual memory layout of the kernel when KASLR is active. A malicious user process accesses a virtual address of the kernel space. The CPU triggers a page fault to the kernel, which then sends a `SIGSEGV` signal to the user process [50, p. 393]. If the page is mapped at this point, the CPU creates a TLB entry for the virtual address under attack. The user process then triggers a page fault on the same virtual address again, and measures the timing it takes to return to the signal handler. If the TLB entry was already present, the fault handling is significantly faster than if the TLB entry has to be resolved again. The attacker can scan the entire kernel virtual address space, and apply a differential analysis on the observed timings to get the kernel virtual memory layout.

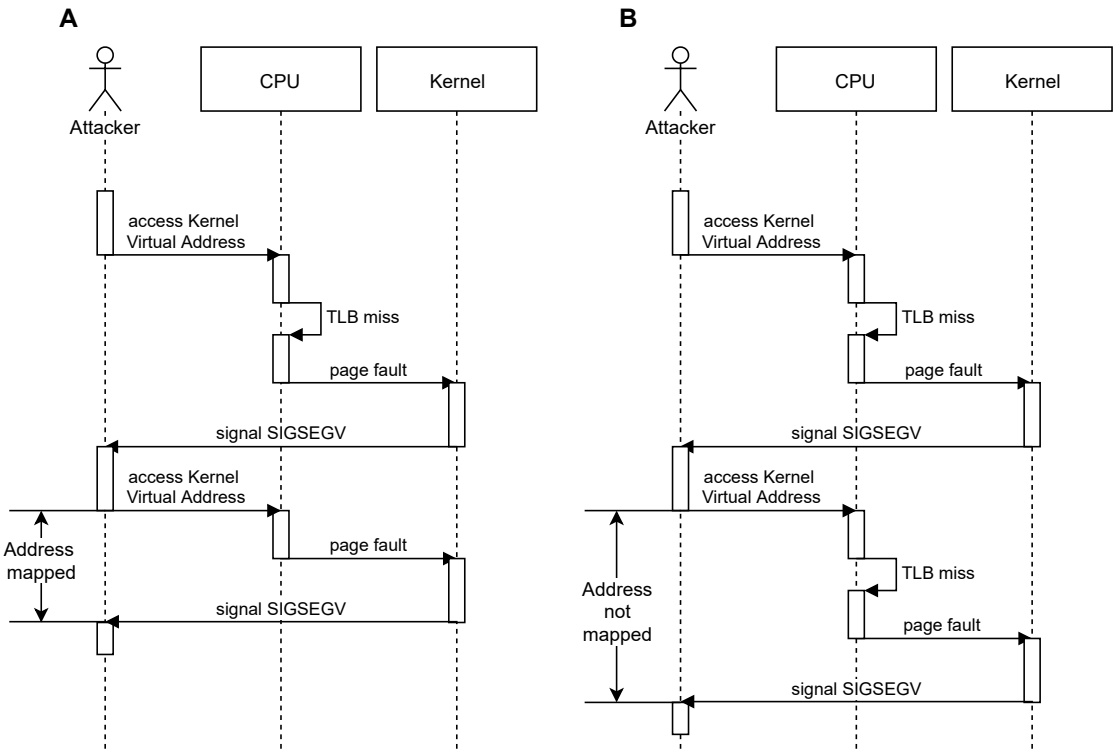

Figure 2.10: An attacker that makes use of the double page-fault side channel [37] first accesses a page of interest, so the CPU attempts to resolve the virtual address to check the access permissions. If the page is present in virtual memory as shown in **A**, the Intel x86_64 CPU updates a TLB entry before the page fault is triggered due to the access permission violation. In the second access on the same virtual memory address, the TLB entry is already valid, and no TLB miss must be handled. For example **B**, the virtual address is not mapped. The CPU does not update the TLB entry upon the first memory access. When the attacker accesses the virtual address a second time, the CPU has to resolve the virtual address again. The attacker measures the time from the second address access to the second signal handler call. If the page is not present as in example **B**, the access takes significantly longer than in example **A**.

Intel TSX-based Side-Channel Attacks. Jang et al. [46] developed the De-Randomize Kernel address space (DrK) attack to break KASLR. DrK is a virtual memory side-channel attack that makes use of the Intel TSX mechanism. When a thread is in an RTM region, and it accesses or executes virtual memory that is not mapped or accessed despite a lack of privileges, Intel TSX aborts the region and

rolls back the state. The attacker cannot directly observe if the virtual memory address is mapped or not. When a thread in an RTM region accesses a virtual memory address, the CPU has to resolve the state of the page, and check if the access is permitted. Due to the multi-stage pipeline and the cache structures, the address resolution and permission checks differ in time until the RTM abort is triggered. The DrK attack measures time, enters a TSX RTM region, and accesses a kernel memory address. The CPU detects that the virtual page is not mapped or requires higher access privileges, and aborts the TSX region. The attacker thread jumps into the abort handler and measures the time again. Depending on the timing difference, the attacker can find out if the page is mapped or not. If the page is mapped, the attacker performs the attack again but executes the address instead of reading it. Again, the attacker measures the time difference, which is smaller if the page is executable. DrK can scan the entire virtual address space (page-wise) of the kernel with this method and fully recover the kernel address space layout.

BTB Attacks. The BTB, as described in section 2.1, is a hardware feature to speed up unconditional and conditional branches by storing the source and target address of branches. Because the BTB is not big enough to store all branch sources, a part of the source instruction address is taken as the index to the BTB-entry, and other parts are taken for the tag. In case a branch occurs and a valid branch is already placed within the BTB-entry, the old entry is overwritten. When the old branch is performed a second time, it is significantly slower than it would be with a valid BTB-entry. This branch timing can be measured and used for timing attacks, as Acıiçmez et al. [3] and [4] have demonstrated. Also, not all parts of an address are used for index and tag, as [20] showed for the Intel Haswell architecture. They found out that only 30 of the 48 address bits are used for the index and tag. Thus, BTB entries may mismatch, even though the index and tag match, leading to a wrong prediction. BTB mismatches are not problematic for the BTB mechanism. In this case, the CPU flushes the pipeline, as it would normally be a BTB miss. However, Evtyushkin et al. [20] used this collision to recover the random bits of the KASLR. As said before, Linux only uses 9 bits of the virtual address to calculate the text section offset at boot time. It randomizes the bits 21 to 29, which is the PD index part of the virtual address. This part of the virtual address overlaps with the index and tag of the BTB entry. The attacker knows branch sources for the Linux kernel. The attacker creates a binary with branch sources at addresses that have a collision with the BTB entry of the kernel branches. Then, the attacker iterates over the possible KASLR offsets. For every possible offset, the attacker uses the branches to fill the BTB entries and then calls a syscall to switch to kernel mode. After the syscall returns, the attacker tries to do the same branch again

and measures its time. If the call takes longer than usual, one can assume that the correct offset for the KASLR text section has been found. BTB attacks only work while running on the same core, but this is irrelevant since the kernel text section addresses are the same on every core of the CPU.

Chapter 3

Kernel Virtual Memory Isolation

This chapter presents software-based countermeasures to mitigate kernel virtual memory side channels. We analyze if they are practical and check if they can impede kernel virtual memory side-channel attacks. At last, we design a countermeasure based on the evaluation and our design goals.

Assumptions. We evaluate countermeasures against the following attack scenario:

[**Assumption A**] The operating system under attack is executed on a non-virtualized x86_64 SMP CPU without PCIDs.

[**Assumption B**] The operating system performed KASLR at boot-time and uses SMAP and SMEP.

[**Assumption C**] A malicious user program wants to perform a ROP attack on the kernel and therefore needs to know the layout of the kernel virtual memory.

[**Assumption D**] The attacker performs a virtual memory timing attack to break KASLR.

Timing attacks on KASLR (as described in Section 2.4) attack the paging structure of the virtual memory, the TLB, or the paging-structure caches [28, 37, 46]. They need to have the virtual kernel memory to be mapped in user mode, the TLB and paging-structure caches to be filled with kernel page entries, and a way to measure time differences.

3.1 Countermeasures

There are different approaches to counter these types of attacks in software. We analyze which countermeasures are practical and check if they can impede kernel virtual memory side-channel attacks.

Disable Time Stamp Counter. All time-based side-channel attacks need a way to measure timing differences. The time stamp counter provides a precise time measurement for attackers. Therefore, many attacks rely on the `RDTSC` instruction to perform their attacks. Hund et al. [37] and Percival [84] suggest disabling the use of the `RDTSC` instruction in x86_64 programs. The Time Stamp Disable (TSD) bit of the CR4 register can restrict the `RDTSC` and `RDTSCP` instruction to privilege level 0 [41, ch. 2.5]. Hund et al. [37] point out that many applications make use of the `RDTSC` instruction, and removing it is impractical. Percival [84] suggests to use this option only on non-SMP systems due to the potential use of Virtual Time Stamp Counters (VTSCs). To implement a VTSC, an attacker can count up a shared variable in a thread on another core. Martin et al. [77] present a method to mitigate VTSC based attacks. Unfortunately, a practical implementation of this countermeasure requires micro-architectural changes.

Manipulate Time Stamp Counter. Martin et al. [77] propose to obscure the behavior of the time stamp counter. The kernel can disable the `RDTSC` instruction, and virtualize the instruction when the privilege violation exception raises. The virtualized `RDTSC` instruction then returns a slightly modified value, which adds entropy to the counter, and makes it unusable for precise time measurements.

Kernel Memory Isolation. An effective way to stop leaking information to attackers is to remove the kernel virtual memory when a thread is user mode. The side channel still exists, but an attacker cannot observe valuable information through it. A thread can either remove or replace the kernel virtual memory when it enters user mode to hide the kernel virtual address layout from side-channel attacks. All TLB entries and kernel space paging-structure cache entries have to be invalidated as well. When the thread enters the kernel again, it has to restore the kernel virtual memory layout.

A problem with this approach is the switch between user mode and kernel mode of a task. If the kernel address space is removed or replaced, the stacks and context switch routines are unavailable when entering the kernel mode. Also, some virtual address regions need to be mapped in user mode because the CPU or user threads

still need them. Therefore, parts of the KVM still have to be mapped in user mode. Figure 3.1 shows the Virtual Memory (VM) map of a process in user mode and kernel mode. The virtual memory mapping in user mode is called Shadow Memory Mapping in this thesis.

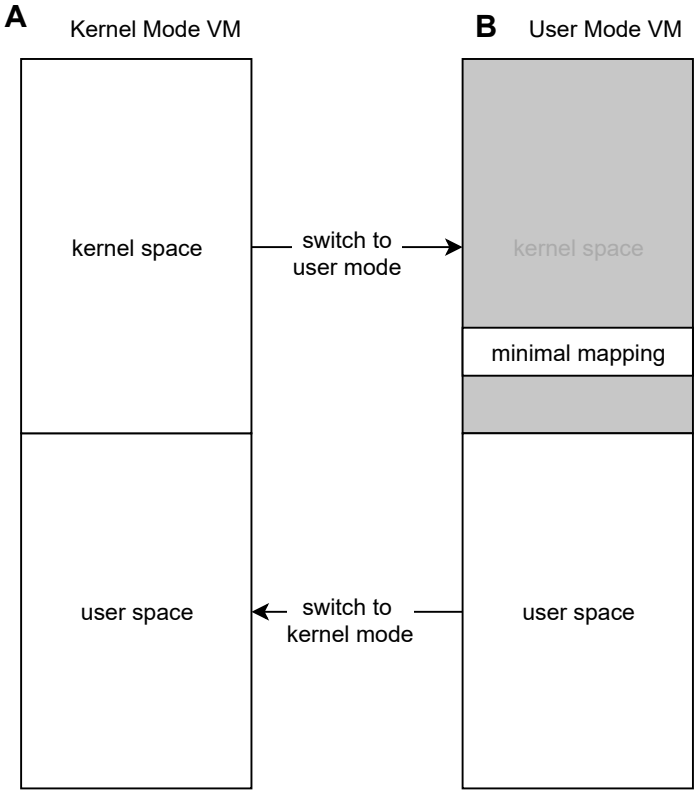

Figure 3.1: The virtual memory layout of a process in kernel mode and user mode. When a thread is in kernel mode (**A**), the full address space is mapped. When the thread is in user mode (**B**), the kernel space gets mostly unmapped. Only some sections still need to be present in user mode.

There are two different approaches to hide the kernel virtual address layout in user mode.

Paging Structure Update . One method to isolate the KVM, is to unmap all kernel pages from the page table upon a context switch to user mode. Gens et al. [23] developed LAZARUS, a technique and Linux kernel patch that implement this countermeasure. LAZARUS only maps the pages the CPU and the kernel need to

have in user mode. Upon a context switch to user mode, LAZARUS replaces the rest of the KVM mapping with a dummy mapping. The dummy mapping does not contain valuable information. When the thread switches to kernel mode, the kernel entry handler restores the kernel mapping upon entry. Figure 3.2 shows an example of a process page table structure in kernel mode and user mode. Since the CPU does not implicitly invalidate the TLB and the paging-structure caches upon a paging-structure update, the kernel has to manually disable the entries that map the kernel virtual memory via the `INVLPG` instruction [41, ch. 4.10.4.1]. LAZARUS does not need to invalidate the minimal kernel mapping and user space TLB and paging-structure cache entries.

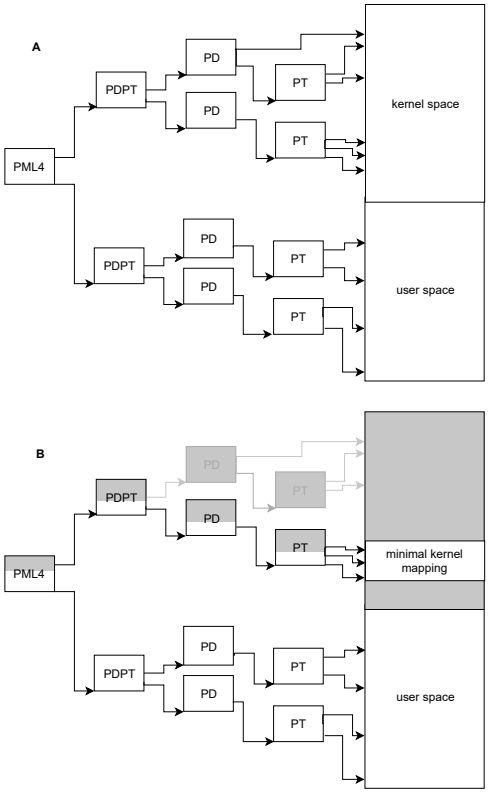

Figure 3.2: In kernel mode (**A**), the paging structure maps the entire kernel and user virtual memory. In user mode (**B**), the paging structure only maps the minimal virtual memory required. The other pages are either unmapped or reference a dummy mapping.

The paging structure update does not work on multi-threaded SMP systems without restrictions. When a thread switches from kernel mode into user mode, the KVM of all other threads within the same process is removed. Therefore,

no other thread can use the same paging structure while one thread is in kernel mode on another CPU core. The kernel can handle the SMP constraints by not scheduling two user threads with the same paging structure on different cores at the same time.

Paging Structure Replacement. Gruss et al. [28] propose *Strong Kernel Isolation* as a countermeasure against the prefetch attack. The paging structure replacement is similar to the 4G/4G VM split patch for the x86_32 Linux kernel [80]. Every user process has two page table structures, one for the kernel mode, and a shadow page table structure for the user mode. Upon switching from kernel mode to user mode, the kernel exit routine replaces the mapping with a shadow mapping. It does so by changing the top-level page table register (CR3). When the thread switches back to kernel mode, the paging structure register is changed to point to the full page table structure again. Figure 3.2 gives an example of the paging structures of two processes. When the CR3 register is changed, the CPU implicitly invalidates all TLB and paging-structure caches except for paging-structure entries marked as global [41, ch. 4.10.4.1]. To invalidate all kernel paging-structure caches and TLB entries when a thread switches to user mode, the kernel needs to remove the global flag from the kernel virtual address space.

Countermeasure Evaluation. Disabling or manipulating the time stamp counter is no practical countermeasure against attacks on SMP systems. Attackers can run a VTSC on another thread or process, and not use the `RDTSC` instruction.

The paging structure update countermeasure does not need to invalidate all TLB and paging-structure caches upon a context switch between user mode and kernel. However, it needs to manually invalidate all kernel TLB and paging-structure cache entries when it enters user mode. The kernel scheduling policy has to be adapted, so no thread in user mode ever shares its paging structure with a thread in kernel mode. Therefore, on a multi-threaded SMP system, the page structure update countermeasure is impractical.

The paging structure replacement countermeasure invalidates all TLB entries and paging-structure cache entries that are not global upon every kernel entry and exit. The option to keep the kernel pages marked as global, and manually invalidate the TLB reduces the kernel page misses but adds additional complexity to the kernel exit functions. Paging structure replacement works on multi-threaded SMP systems without any additional scheduling policies. Threads in kernel mode and threads in user mode never share their paging structure by design.

3.2 Design of KAISER.

In this thesis, we present KAISER, a minimalistic countermeasure against attacks on the kernel virtual memory from user programs. We designed KAISER to achieve specific goals. KAISER shall have minimal impact on the structure of the operating system. KAISER shall be easy to analyze for potential new bugs or side channels. KAISER shall be portable to other operating systems or CPU architectures. KAISER shall impede a class of side-channel attacks. KAISER shall have a low boot time impact. KAISER shall have a low static runtime memory overhead. KAISER shall have a low per-thread and per-process memory overhead. KAISER shall have a low runtime overhead.

KAISER implements a paging structure replacement mechanism to hide the kernel pages in user mode. It creates a shadow paging structure for every process. The shadow paging structure maps only the necessary minimal virtual addresses for CPU features. All other virtual kernel pages are not present.

Upon boot, KAISER sets up a single initial kernel shadow mapping. New processes copy the shadow mapping kernel part from the current process or kernel thread. Processes share the shadow paging structure tables except for the top-level page tables, so a new process just needs to reserve a new top-level page table. Figure 3.3 shows how two processes share their paging structures between each other and between the full mapping and shadow mapping. Whenever the kernel modifies the virtual memory layout of the user space, KAISER immediately synchronizes it with the shadow memory.

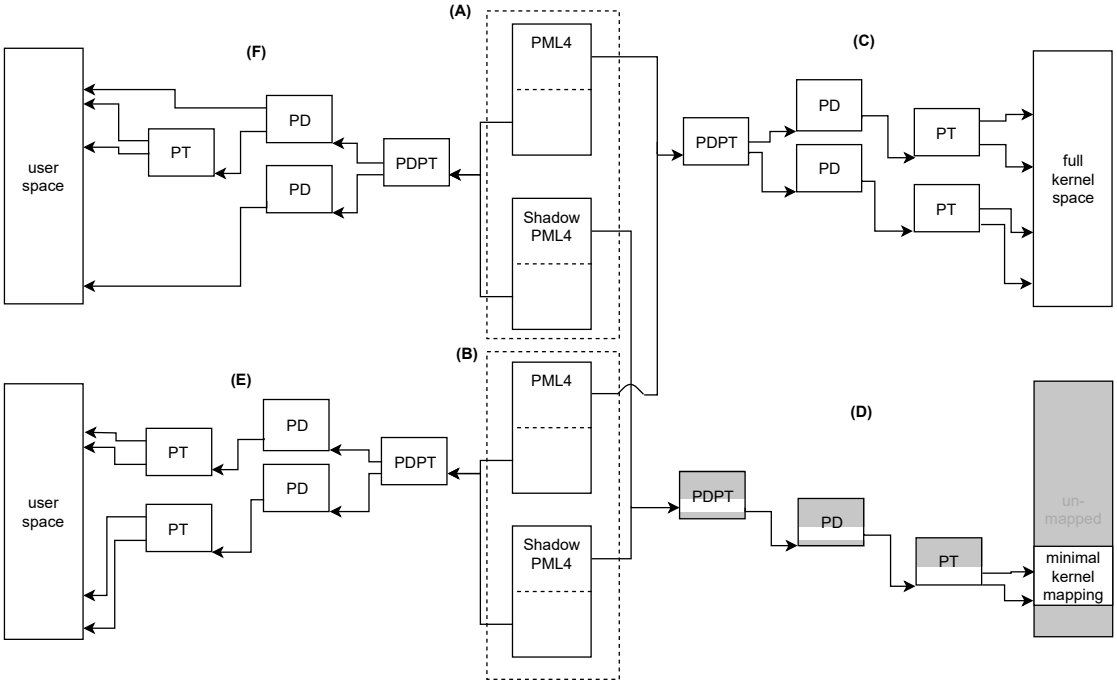

Figure 3.3: KAISER extends the paging structure of processes (**A**) (**B**) by a shadow top-level page structure. In this case, the system supports four-level paging. Every process gets an additional shadow PML4 4 for the user mode of threads. Modern operating systems share the kernel paging-structure tables (**C**) between processes to reduce the overhead of paging structure tables and synchronization. KAISER also shares the shadow kernel paging structures (**D**) between processes for the same reason. A process also shares the userspace paging structure tables (**E**)(**F**) between the full paging structure and the shadow paging structures. The PML4 4 and shadow PML4 4 user parts are always in sync.

When the kernel creates a new user thread, it maps the thread's kernel stack into the shadow mapping. When the kernel deletes the thread, it removes the kernel stack mapping again.

Whenever a thread exits the kernel to user mode, KAISER switches to the shadow mapping. This switch happens just before the return instruction, so the shadow memory has to map only the minimally necessary virtual memory areas. Upon a kernel entry from user mode, the current thread immediately switches to the full paging structure. Entry gates for interrupts and exceptions can be called in both kernel and user mode. To avoid paging structure switches when

possible, interrupt and exception handlers check whether the current thread came from kernel mode. If the interrupt or exception was triggered while the thread was in kernel mode, it does not need to change the paging structure. Upon an exit from an interrupt or exception, the thread switches to the shadow mapping only if it is returning to user mode. A thread shall never return from kernel to kernel mode with the shadow mapping, as this would trigger a page fault because the text section of the kernel is not present.

To ensure that all TLB entries and paging-structure caches are invalidated when a thread is in user mode, KAISER removes the use of global mappings. When the kernel switches a paging structure, the CPU implicitly invalidates all paging-structure caches and TLB entries. The overhead introduced by KAISER through disabling the global bit of the paging structures only affects kernel pages. User processes do not share the same virtual address-space layout and, thus, their pages are not globally mapped.

Chapter 4

Implementation in Linux

This chapter focuses on the KAISER proof-of-concept kernel patch. We show that it is feasible to implement a kernel protection mechanism that can hide the kernel virtual address layout in user mode and hinder side-channel attacks. KAISER is implemented in Linux kernel version 4.10 for x86_64 platforms with default settings. The KAISER patch runs on a non-virtualized SMP CPU with four-level paging. It can be enabled at compile-time via the Linux Kconfig system. In the last section of this chapter, we analyze the boot-time and run-time performance and memory impact of KAISER.

4.1 Shadow Memory

As explained in Section 3, the Linux kernel memory needs to be hidden in user mode. Linux and x86_64 CPU long-mode mechanisms still require some kernel virtual memory areas to be present in user mode. KAISER implements a mechanism to copy regions from the kernel to an initial shadow mapping and create new paging structure tables. The initial shadow memory PML4 4 page is reserved in `head_64.S`. The kernel sets up the initial shadow memory PML4 4 after the memory layout, but before it sets up the scheduler and creates the first user processes.

Entry and Exit handlers. The entry and exit points of the Linux kernel have been described in Section 2.2. The kernel linker scripts already place all entry and exit points into the `.entry.text` section, and define labels for the entry section start and end. Since this section only contains the minimal entry and exit functionality, KAISER fully maps the entry section into the shadow memory.

Stacks. Some kernel exception and interrupt entries and exits need a stack to push or pop data. The Linux kernel has different stacks per core - one IRQ stack, four different exception stacks, and the ESP fixup stacks, which are described in Section 2.2. The CPU uses them upon a switch between user and kernel mode, so KAISER maps them into the shadow memory. The Linux kernel already maps the ESP fixup stacks into an exclusive PUD range. KAISER directly references the ESP fixup PUD to save physical memory.

vsyscalls. The deprecated `vsyscall` mechanism allows a user program to directly call functions located at the top of the virtual memory. The x86_64 Linux kernel does not map an executable section to the `vsyscall` region. Instead, it handles `vsyscall` calls in the page-fault handler. Therefore, the `vsyscall` area does not need to be mapped in the shadow mapping.

Descriptor Tables and TSS. The descriptor table registers contain the virtual base addresses of their tables. x86_64 also requires a TSS to be set up per core. The CPU needs the TSS, IDT, and GDT to be present all the time. Since their location and size do not change post-boot, KAISER only needs to map them once.

Context Switch Variables. To exchange the CR3 register, the entry and exit code need variables to switch between kernel and user mode. These variables are per-CPU defined. Their functionality is described in Section 4.2.

Per-CPU Memory. The Linux kernel handles the indexing of the per-core variables via the GS, which includes an offset within a per-CPU virtual memory section. KAISER splits the layout into a user-mapped and a non-user-mapped area. The per-CPU memory is located at the beginning of each per-CPU region within the section. Only these regions of the per-CPU section are virtually mapped while a thread is in user-mode.

Run-time Mapping. The Linux kernel creates user threads at boot time, and via the `fork` mechanism. Each thread has its own kernel stack. System calls and interrupts use the thread stack to enter and exit, so the thread stack has to be mapped in user-mode. Whenever the kernel creates a new thread, it clones the current thread's `task_struct`, and allocates new stack pages. KAISER maps the allocated stack into the shadow mapping. When the kernel frees the thread stack, KAISER removes it from the mapping. These stacks are allocated page-aligned

in the `vmalloc` region. Because the kernel modifies the shadow memory paging structure at several points, the shadow page tables have a shared spin lock, that a thread acquired whenever it adds or removes a virtual memory area in the shadow memory.

Global Pages. Upon a switch to user mode, KAISER has to ensure that all kernel TLB and paging structures are empty. Linux marks kernel pages like the kernel executable pages as global in the paging structure table entries. The references to these pages will stay in the paging structure caches and the TLB when the CR3 register is updated. To completely invalidate the kernel entries, KAISER sets the `_PAGE_GLOBAL` mask in `arch/x86/include/asm/pgtable_types.h` to 0, and effectively disabling the global bit for all memory.

Kernel-Space Synchronization. The kernel part of the shadow PML4 4 needs to be synchronized between all threads. Since the kernel adds and removes kernel thread stacks in the shadow mapping at run-time, the shadow paging structure can contain duplicated and non-synchronized tables. Since a thread stack is cleaned up by a thread other than its owner, it has to have the same shadow kernel PGD entries as the cleaned-up thread. The kernel could always iterate over all shadow PML4 4s, and synchronize them whenever the shadow memory layout changes. KAISER instead directly maps the kernel PUD pages into the shadow memory at boot. This way, all PGDs always have the same layout and do not need further kernel space synchronization.

User Space Synchronization. All user threads of a process share the same virtual user memory. The kernel has several places in which it needs to copy data between kernel space and user space. The user mapping stays present in kernel mode. KAISER extends the function `native_set_pgd(pgd_t *pgdp, pgd_t pgd)` to synchronize the user space part of the full mapping and the shadow mapping. A PGD has to be page-aligned, so a PGD entry references user space if `((unsigned long)pgdp % PAGE_SIZE) < (PAGE_SIZE/2)`. Whenever the kernel sets a PGD entry in user space, KAISER copies the content it into the corresponding shadow PGD entry. Kernel modules and functions are not allowed to exit the kernel on their own. They need to use entry-section functions. To ensure that a program does not use an alternative kernel exit point, `native_set_pgd(pgd_t *pgdp, pgd_t pgd)` disables the user flag in the shadow user-space PGD entries. If a kernel thread would exit to the user-mode via an alternative exit point, the Memory Management Unit (MMU) triggers a protection fault. This mechanism also ensures that KAISER does not skip any valid exit points.

PGD Entry Conversion. Linux accesses the paging structure entries via the physical direct mapping. KAISER needs a mechanism to convert PGD entries to shadow PGD entries and back: KAISER puts the shadow PGD on the next physical page of the full PGD. It also aligns the PGD physical address to `2 * sizeof(PAGE_SIZE)`. The kernel accesses the PGD entries via the direct physical mapping, so the placement is both useful for virtual addressing as for CR3 updates. The following snippet shows how KAISER performs the conversion between full and shadow PGD:

```
USER_CR3 = CR3 | PAGE_SIZE
KERNEL_CR3 = CR3 & ~PAGE_SIZE
pgd_t* shadow_pgdp = &pgdp[512]
```

The CR3 conversion shown above is fail-safe: A thread can switch to the full or the shadow PGD based on the current CR3 value. If the kernel would compute the CR3 value by an offset, and a bug causes the kernel to switch to user mode or kernel mode twice, the CR3 does not refer to the correct PGD any longer. By using a mask, KAISER ensures that double called entry and exit functions (e.g., via an NMI) do not cause the address resolution to break.

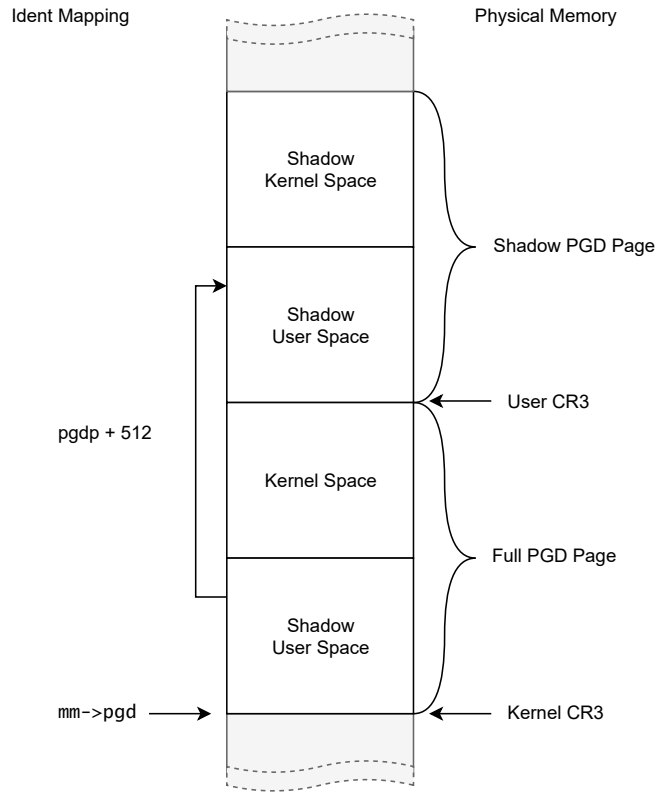

Figure 4.1: KAISER puts the shadow PGD on the physical page next to the full PGD, and aligns the physical full PGD address to `2*PAGE_SIZE`. Using this trick, KAISER can resolve the shadow PGD entry of any full PGD entry. It converts a PGD entry pointer to a shadow PGD entry pointer by adding `PAGE_SIZE` to them. The `mm_struct` holds the reference to the full PGD.

4.2 Kernel Entry and Exit

The kernel entry and exit handlers, as described in 2.2, have to switch the GS as soon as they switch between kernel and user mode. KAISER has to switch the CR3 as soon as possible because some entry and exit points access global per-CPU variables. Also, some ISR entry and exit points don't have a clean stack to enter or exit. Therefore, KAISER implements macros to switch to the kernel CR3 and the user CR3 of the current thread:

```
.macro _SWITCH_TO_KERNEL_CR3 reg
movq %cr3, \reg
```

```
andq $(~0x1000), \reg
movq \reg, %cr3
.endm

.macro _SWITCH_TO_USER_CR3 reg
movq %cr3, \reg
orq $(0x1000), \reg
movq \reg, %cr3
.endm
```

The CR3 register is only accessible by `MOV` instructions [41, ch. 2.5], so KAISER needs an additional register to modify it. Most entries and exits push the temporary register content onto the stack. Unfortunately, some exceptions and interrupts have no save-to-use stack available when they enter or exit an ISR (e.g., the NMI handler). For these handlers, KAISER reserves a register backup memory in the user mapped per-CPU variables.

The kernel has to always execute the `swapgs` instruction when the current thread switches from kernel to user mode and from user to kernel mode. KAISER makes use of the existing entry and exit checks. It switches to the kernel PGD right after the entry `swapgs` instruction, and to user PGD right before the exit `swapgs` instruction.

4.3 Performance and Memory Impact

We design the KAISER implementation to be memory and performance efficient. In this section, we have a look at the performance and memory overhead introduced by the KAISER patch. We have a look at the static and dynamic physical memory overhead and performance impact. Then, we present benchmarks of a Linux operating system with KAISER to see the performance impact on a real system.

Cache Performance Impact. KAISER updates the CR3 register on every kernel entry from user mode or exit to user mode. In addition, it marks all virtual kernel pages as non-global. As a result, every kernel entry and exit, and every change of the current PGD, invalidates all TLB and paging-structure cache entries. Figure 4.2 shows transitions of threads between user mode and kernel mode on a single core. It highlights when the kernel updates the CR3 register.

The scheduler is responsible for transitions between user-mode threads. Since

Linux handles thread scheduling in kernel mode, every transition between user-mode threads updates the CR3 register at least twice. If the current user-mode thread switches to a user-mode thread of another process, the kernel updates the CR3 three times. The first CR3 update already invalidates all cache entries. The following CR3 updates do not impact the performance as much, because the user- and kernel TLB and paging structure cache entries are already invalid.

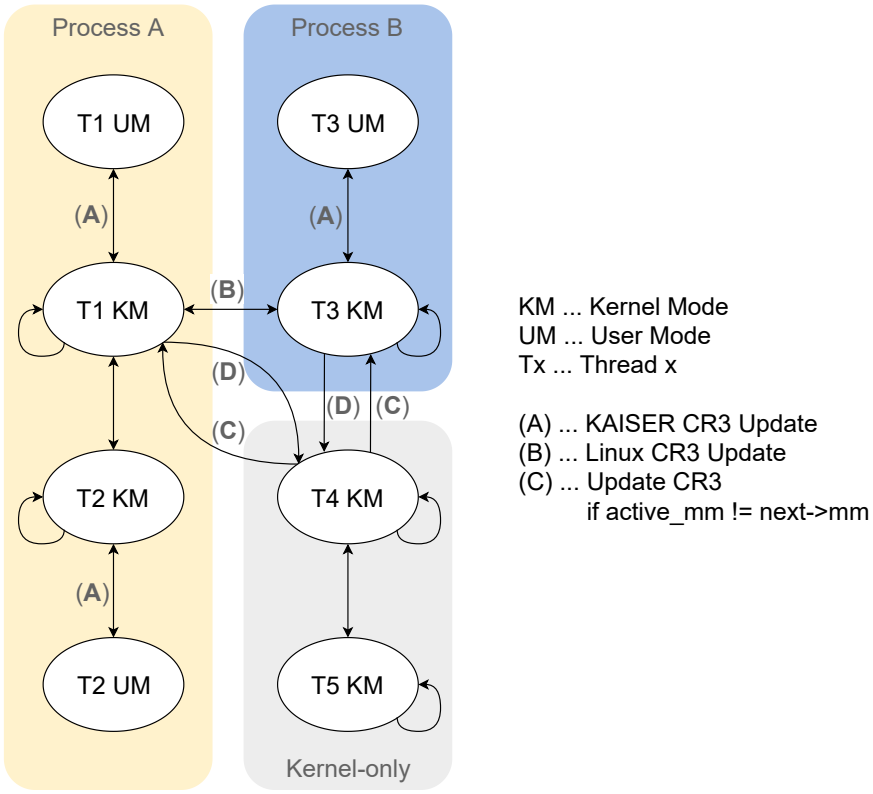

Figure 4.2: KAISER adds additional CR3 switches and invalidates TLB and paging structure cache entries. This example highlights the CR3 switches between different threads. When a user thread switches between user mode and kernel mode (`A`), KAISER swaps the CR3 value between full PGD and shadow PGD. The scheduler changes the CR3 register when it switches between kernel threads of different user processes (`B`). If a user thread in kernel mode switches to a kernel-only thread (`D`), it keeps the current virtual address space. When the scheduler switches from a kernel-only thread to a user thread in kernel mode (`C`), it needs to update CR3 if the virtual address space is not identical. Other context switches, or kernel entry and exit points, do not require to switch the PGD at run-time.

Computational Overhead. KAISER has low initialization and run-time computational overhead. At boot time, KAISER just needs to create shadow page tables, and link them to the full mapping pages. Whenever the kernel creates a new process, KAISER allocates two pages and copies the PGD entries to the shadow mapping. When the kernel creates or deletes a user thread, KAISER has to update the shared shadow mapping structures and therefore acquire a global shadow table lock. The kernel synchronizes all other accesses to the PGD and shadow PGD page by `mm_struct->page_table_lock`. Whenever a thread changes a user-space PGD entry, KAISER clones it into the shadow PGD entry via a simple pointer update. The KAISER entry and exit CR3 switch only needs three to five additional instructions per CR3 update. As shown in Section 4.2, it requires only three instructions (plus an unused register) to change from user- to kernel-CR3 . To sum it up, the boot and static KAISER setup are very cheap computation-wise compared to the CR3 update. The most significant run-time computational overhead is the update of the shadow paging-structure. Whenever the kernel creates or removes a user-thread, it locks the shadow paging structure for all CPUs and acquires or releases up to 2 pages for the page tables.

Performance Measurements. In our paper about KAISER [30], we evaluated the performance of the KAISER proof-of-concept path by benchmarking it. We ran different benchmarks with and without KAISER on an Intel Core i7-6700K Skylake CPU with 16GB of RAM. The result is presented in Figure 4.3. As benchmarks, we chose pgbench [95], PARSEC 3.0 [85], and SPLASH-2-x [85]. The execution time for the benchmarks with KAISER was compared to the execution time without KAISER, yielding the execution overhead. We performed the measurements for 1, 2, 4, and 8 threads on different cores to see the impact of KAISER on a multi-threaded SMP system.

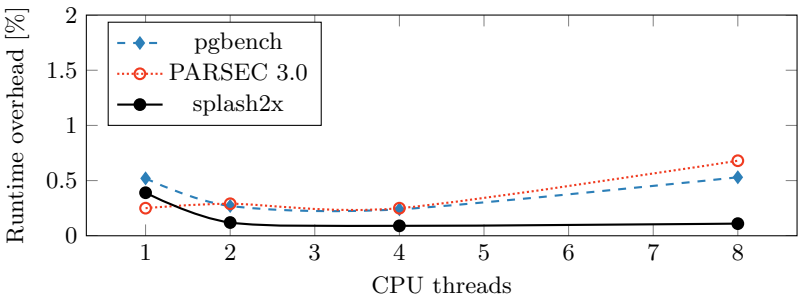

Figure 4.3: Gruss, Lipp, Schwarz, Fellner, Maurice, and Mangard [30] benchmarked the Linux KAISER patch on an Intel Core i7-6700K. We compare their results against the same kernel without KAISER. This graph shows the overhead measured by different benchmarks over a different number of threads. The performance overhead is 0.28 % on average, with a peak overhead of 0.68 %.

Fogh [21] estimates that a countermeasure like KAISER would lower the system performance by around 5 %. Gruss et al. [30] speculates that the much lower performance impact of only 0.28 % on average comes from undocumented Intel features. The Intel manuals state that the CPU invalidates all non-global TLB and paging-structure cache entries upon a CR3 switch [41, ch. 4.10.4.1]. Gruss et al. [30], however, speculate that Intel CPUs like the Intel Core i7-6700K internally tag TLB and paging-structure caches, e.g., , with the CR3 register value [99]. Thus, similar to the PCID mechanism, the CPU would not invalidate the entries upon a CR3 update. This would explain the low performance impact of KAISER observed in benchmarks.

Memory Overhead. KAISER needs additional physical memory, that is reserved in the Linux kernel image, or allocated at boot time. First, KAISER increases the .text section. At boot time, KAISER reserves an additional shadow PGD page next to the initial PGD page. KAISER pre-reserves a PUD page for every kernel shadow mapping. Then, KAISER maps the user per-CPU map, the kernel entry text section, and the IDT into the shadow memory. KAISER maps 13 pages from the per-CPU area into the shadow memory per core. The layout of the user mapped area and their size are listed in Table 4.1.

Linux has a dedicated allocator for kernel and module per-CPU variables. On x86_64, the kernel sequentially places the per-CPU variables into the virtual memory with additional spacing for modules. It reserves `0x80000` bytes (or 128 pages) per core. The kernel aligns the first per-CPU area with beginning at the first

Name	Size
IRQ stack	16 kB
TSS and I/O bitmap	102 B + 8 kB
GDT page	4 kB
exception stacks	3 × 4 kB + 8 kB = 20 kB
register_backup	8 B
pages/core	13 × 4 kB pages

Table 4.1: The user-mapped per-CPU memory and the number of virtual user-mapped pages per core.

entry of a PMD [65]. KAISER, therefore, needs to allocate one PT page per 4 cores (rounded up) and one PMD page. The number of CPUs is limited to 64 per default, and one PMD can references to up to 512 PUDs, and up to 2048 per-CPU sections.

Whenever the kernel creates a new process, it allocates one page for the PGD. With KAISER, it allocates two physical pages instead. KAISER maps the 16 kB stack of a user-thread into the shadow memory. Thus, it needs two pages for PT and PMD. The number of stack mapping pages may vary depending on the alignment of the virtual address because the stack spans over two pages, and can hit a PMD or PT virtual address border. Table 4.2 lists the overall physical memory overhead KAISER introduces to the kernel.

As one can see, the most memory overhead of KAISER is caused by the pre-allocated PUD pages for the shadow memory. The compressed Linux 4.10-rc6 kernel bzImage with KAISER is only 1280 Bytes larger than the same bzImage compiled without KAISER.

Name	Size
.text Overhead	5 kB
entry & exit mapping	2 pages
per-CPU PMD	1 page
per-CPU PTs	1 page / 4 CPUs
IDT Mapping	2 pages
initial Shadow PGD	1 page
process Shadow PGD	1 page / process
thread stack mapping	2 pages / thread
Shadow PUDs	255 pages
overall overhead	1.02 MB + 4 kB/4CPUs + 4 kB/process + 8 kB/user thread

Table 4.2: The additional KAISER memory overhead depends on the number of cores, number of processes, and the number of user-threads. To measure the `.text`, `.bss`, and `.data` section overhead, we compiled the Linux kernel with gcc 5.4.0, and compared the size of the sections with and without KAISER. The result can vary depending on the configuration of the kernel, drivers compiled with the kernel, the position of the thread stacks. It also does not include the overhead of physical page management.

Chapter 5

Security Evaluation

While we have shown that KAISER is an efficient patch with a memory overhead of ≈10 MB and performance overhead of ≈0.28 % on Intel Skylake CPUs [30]. In this chapter, we have a look at the security impact of KAISER. First, we look at different cache side-channel attacks and evaluate if KAISER can mitigate them by design. We present measurements of side-channel-attacks on the KAISER proof-of-concept implementation and compare them to attacks on a kernel without KAISER. In the end, we identify weaknesses of the KAISER patch and how one can fix them.

5.1 Side Channels

The KAISER patch isolates the kernel virtual memory. It mitigates timing leaks in the virtual to physical address resolution of paging. It also ensures that a user thread can obtain no direct timing information of the TLB and paging-structure cache entries that refer to kernel memory.

Whenever a thread switches to user mode, most of the kernel virtual memory is unmapped. Also, all TLB and paging-structure caches are invalidated. Since CPU cores do not share the TLB and paging-structure caches, a thread does not leak kernel information to a user thread that runs on another core.

Flush+Flush, Flush+Reload, and Prime+Probe attacks use side-channels if the L1, L2, and L3 caches [29, 82, 101, 102]. The CPU does not invalidate L1, L2, and L3 (LLC) caches upon a CR3 update. Therefore, KAISER does not invalidate any caches that resolve a physical address to a cached value, potentially leaving certain side channels open.

BTB attacks attack the kernel via timing leaks of the BTB. Intel x86_64 processors implement a Indirect Branch Predictor Barrier (IBPB) to flush the BTB [43, ch. 2.4.3]. KAISER does not use the IBPB upon kernel entry and exit. It flush the BTB, and does not prevent BTB side-channel attacks on the kernel.

5.2 Attacks

In Section 2.4, we presented state-of-the-art timing side-channel attacks on CPU caches. This section evaluates whether KAISER prevents these attacks.

Prefetch Side-Channel Attacks, as explained in Section 2.4, targets the virtual memory and the paging-structure caches of the kernel with a Flush+Reload mechanism [28]. Gruss et al. [28] assumed that the prefetch side-channel attack needs the virtual memory of the kernel to be mapped in the user mode. Consequently, KAISER can prevent this attack. When an attacker performs a translation-level recovery attack from a user-mode thread, the attacker can recover the kernel shadow virtual memory layout. We performed an attack on a Linux kernel hardened with KAISER [30]. Our results are illustrated in Figure 5.1. Schwarzl et al. [90] later showed that the address-translation attack, which is able to recover the virtual to physical memory mapping of the kernel, is actually based on another side channel. They proved that the prefetch instruction itself does not leak any information about the kernel virtual to physical memory mapping. The KAISER patch prevented the address-translation attack by accident. The reason is explained in Section 6.3.1.

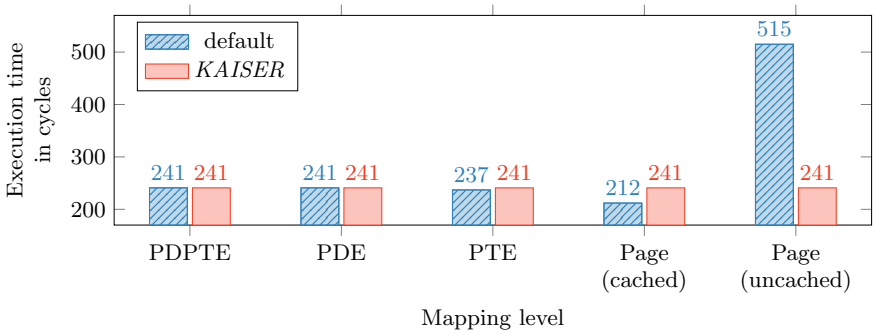

Figure 5.1: We performed a translation-level recovery attack on the kernel virtual memory. Without KAISER, an attacker is able to identify the parts of a virtual kernel address that is preset in the paging-structure and TLB entries. With KAISER, the attack does not leak any timing information about the kernel mapping any longer [30].

The **double page-fault attack**, which we described in Section 2.4, accesses a virtual kernel address twice [37]. The first time, the CPU loads the corresponding TLB entry if the page is present. The second time, the attacking thread measures the time between the access and the `SIGSEGV` signal from the kernel. If the virtual page is present in virtual memory, the CPU does not need to handle a TLB miss and, thus, throws the page fault faster. Since KAISER invalidates the TLB upon every switch from kernel to user mode, the second double page-fault TLB lookup always misses. Figure 5.2 shows an execution time measurement of a double page-fault attack. The kernel mapping does not leak any information with KAISER.

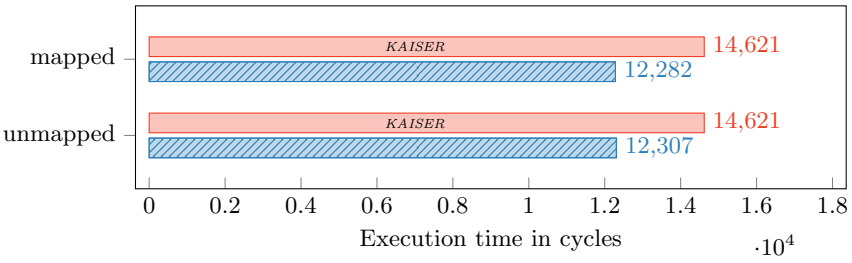

Figure 5.2: The execution time of a double page-fault attack leaks the mapping status of a kernel page. When KAISER is active, the attack cannot distinguish mapped from unmapped kernel pages [30].

The **DrK attack** accesses and execute kernel virtual memory in an Intel TSX RTM region. The CPU detects the privilege violation and aborts the RTM region.

The attacker measures the processing time of the RTM region. Depending on the timing, the attacker can determine the level of the mapping, and whether it is executable. KAISER removes the virtual kernel memory in user mode, so the DrK attack can only see the shadow memory layout. The measurements presented in Figure 5.3 shows that KAISER closes this side channel.

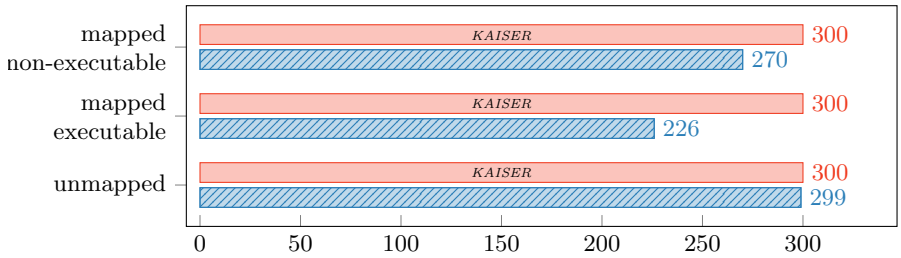

Figure 5.3: Without the KAISER patch, the DrK attack can find out if a page is mapped in kernel space, and if it is executable. With the KAISER patch, an attacker cannot observe the mapping state of a virtual kernel page by the timing any longer Gruss, Lipp, Schwarz, Fellner, Maurice, and Mangard [30].

5.3 Summary

In this section, we have shown that KAISER can prevent certain attacks on the kernel virtual memory layout. The double page fault attack cannot recover any mapping since the kernel invalidates the TLBs every time the attacking thread returns to user mode. With DrK and the prefetch attack, an attacker can only recover a layout of the shadow kernel mapping.

The KAISER proof-of-concept implementation only maps the necessary kernel parts into the virtual memory of the user thread. However, it does not randomize the virtual addresses of the mapped regions. KASLR, as shown in Figure 2.8, aligns the randomized regions at 1 TB, the kernel text section at 2 MB, and the modules at 4 kB. Due to the sparse virtual memory layout of the kernel, an attacker can recover the offset of KASLR regions that the shadow memory maps.

A real-world KAISER patch needs to place all shadow mapped regions at a fixed location in virtual memory like a fixmap to close this side channel. This way, an attacker can see the fixed region layout, but cannot obtain any information about the KASLR randomized regions.

Chapter 6

Practical Impact

In this chapter, we have a look at the practical impact of KAISER. First, we show what happened to the original patch, and how it got merged into the Linux kernel mainline. Then, we present new information leakage attacks: Meltdown [73] and Spectre [52]. These attacks exploit a weakness of out-of-order CPUs to read data from the protected kernel. We describe how they work and evaluate whether KAISER defends the kernel against them.

6.1 KAISER Improvements

We designed and implemented KAISER to prevent attacks on KASLR, and made the proof-of-concept implementation available to the public in the form of a patch[1] [30]. In October 2017, Dave Hansen picked up the KAISER patch. He cleaned it up in order to integrate into the Linux kernel v4.14 [13, 32, 33]. The patch set fixed bugs on the kernel entry and exit functions that crash the kernel under special circumstances. Hansen added trampoline stacks for the kernel threads, so the shadow memory does not need to map all thread stacks and instead can hide the kernel physical page mapping. It also maps LDTs into the shadow memory, since some user programs use the LDT via the `modify_ldt()` syscall [14, 69]. Furthermore, Hansen added support for the PCID feature to lower the performance impact of KAISER. The CPU tags the TLB entries and the paging-structure caches with the current PCID, and does not invalidate them upon a context switch [41, ch. 4.10.1]. In their implementation, the kernel and user processes have different PCIDs. When a thread switches from or to kernel mode, it updates the PCID at the same time

[1] Available at `https://github.com/IAIK/KAISER`.

as the CR3 register. This way, the CPU does not flush the tagged kernel and user TLB entries upon a context switch. Still, an attacker cannot observe information via side channels from user mode, because a user thread can only access the TLB entries with the PCID of its process.

To further decrease the performance impact, the patch set enables the use of the global bit for shadow mapped pages again. Since the kernel maps shadow kernel pages in user mode and kernel mode at the same virtual addresses, they do not leak any additional information about the layout. Furthermore, the patch set changes the paging structure to make extensive use of huge pages. The huge page TLBs entries can handle 2 MB and 1 GB virtual memory pages directly. Thus, huge pages reduce the number of TLB entries that KPTI flushes, and the CPU has to reload.

KAISER was not added into the Linux kernel v4.14 [70]. Instead, kernel developers extended KAISER and renamed it to KPTI [14]. They also added support for Intel's 5-level paging [16], various bug fixes, as well as arm64 support [17], and released KPTI in the Linux kernel v4.15 [71].

6.2 KAISER / KPTI Performance

As we already discussed in Section 4.3, KAISER/ KPTI add a significant performance overhead. Even though [30] benchmarked the performance overhead of KAISER to be on average 0.28 % and 0.68 % in the worst case, [32] estimates it to be 5 % on average and 30 % in the worst case. The operating system performance impact of KAISER/ KPTI depends on the several parameters [26]. Processors have different architectures, TLB, and paging-structure cache sizes, and support different performance improvements like PCID. A higher syscall rate and number of context switches of a user thread increases the number of switches between user mode and kernel mode, as Figure 4.2 illustrates. Interrupts and exceptions, like page faults, in user mode also switch to kernel mode and back. Finally, the working-set size and virtual-memory access pattern of user programs and the kernel also influence the number of TLB and paging-structure cache entries that are in use.

With the performance impact of KAISER/ KPTI depending on multiple factors, Gregg [26] measured an KPTI overhead of −5 % to 800 %, depending on the parameters described above. As the resulting overhead measurements in Figure 6.1 show, hardware acceleration features significantly decrease the KPTI overhead. PCID reduces the impact to 2.6 % at 50 000 syscalls per second. With large pages

and Intel's large page TLBs, the operating system even performs better with KPTI at certain syscall frequencies [31].

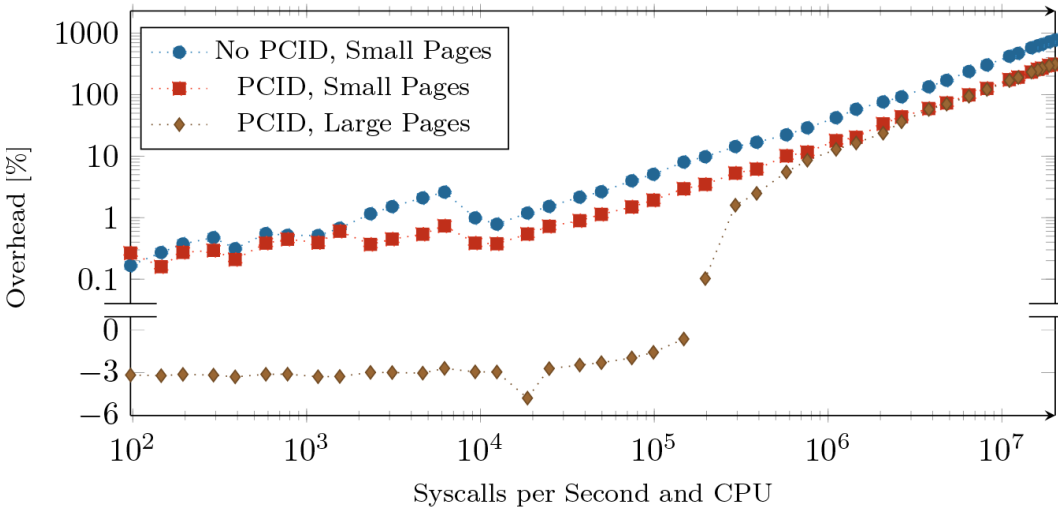

Figure 6.1: This graph shows the KPTI overhead with different optimization methods for different syscall frequencies. The overhead axis is logarithmic above 0 %, and linear below 0 %. With PCID and large pages, a kernel with KPTI can be even faster than a kernel without [31].

6.3 Transient-Execution Attacks

In January 2018, Lipp et al. [73] and Kocher et al. [52] introduced two new sorts of attacks on kernel memory: Meltdown and Spectre. Based on the research of Fogh [22], Meltdown and Spectre bypass privilege isolation, and allow an attacker to obtain protected data from user mode. Both Meltdown and Spectre make use of transient-execution leaks. Transient execution is the erroneous ahead-of-time computation of instructions that the CPU discards [10], as described in Section 2.1. Intel, IBM, Arm, and AMD processors are vulnerable to transient-execution attacks [8, 36, 38, 40, 52].

6.3.1 Spectre

State-of-the-art CPUs, as described in Section 2.1, can speculatively execute instructions of a branch to optimize the system load. The branch predictor computes the most-likely branch target based on previous branch targets. The CPU then

loads and executes instructions of the speculated branch target. Once the CPU resolves the actual branch target, it keeps the results if the prediction is valid, or discards them if it is invalid. Even though the CPU reverts the register and memory state, it does not reset the microarchitectural state. Caches and buffers still hold information about the speculative execution. Spectre attacks recover this information leak. They can thereby observe information, and bypass privilege checks. The types of Spectre attacks are shown in Figure 6.2.

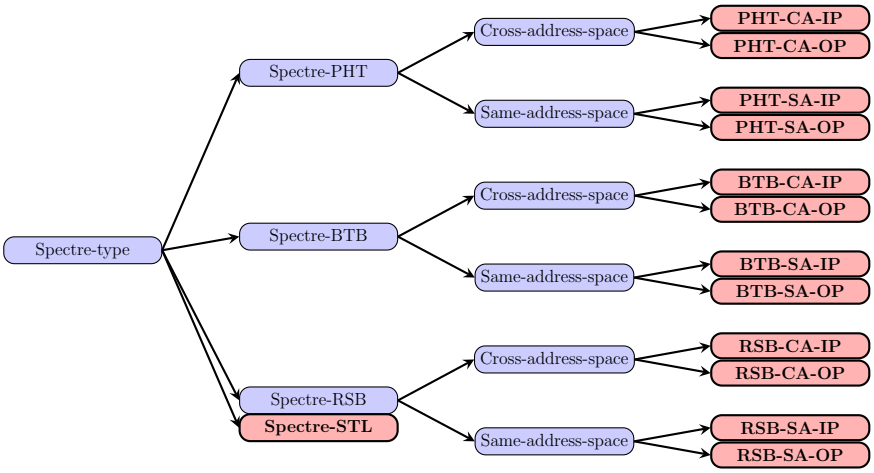

Figure 6.2: This figure visualizes the different variants of Spectre attacks. The non-leaf nodes of the graph represent attack groups, and the leaf nodes are actual attacks [10, 81].

To give an idea of a Spectre attack, we showcase a Spectre-Pattern History Table (PHT) attack based on the code snippet in Listing 6.1. In this example, the attacker is in control of the byte index `x`, has access to the byte array `array2`, and can trigger the victim to execute the if-clause (e.g., via a syscall). First, the attacking thread flushes `array2`. Then, the attacker trains the branch predictor. It triggers the snippet execution with a valid value for `x` until the branch predictor consistently predicts to branch into the if-clause. Then, the attacker sets `x = &secret - &array1[0]`. When the victim executes the snippet, the CPU speculatively reads the value `array2[secret * PAGE_SIZE]` from the cache. It validates the corresponding page in the data caches before it discards the speculative results. The attacker then recovers the secret by checking for cached pages: `secret = argmin(z, read_access_time(&buffer2[z * PAGE_SIZE]))`. Be aware that the training phase already caches the `array2` pages. The attacker either filters for the trained entries or flushes `array2` after the branch prediction training phase.

```
if (x < array1_size) {
    /* The CPU can speculatively execute the array
       access, even if the boundary check fails. */
    y = array2[array1[x] * 4096];
}
```

Listing 6.1: A Spectre-PHT in-place (Spectre PHT-SA-IP) conditional branch snippet, which is executed by the a thread in kernel-mode, or a thread of another process [52].

As shown in this example, the attacking thread does not need to have the secret mapped, nor does it use caches or buffers that the CPU invalidates upon a CR3 update. The attacker can use a syscall to validate the data cache entry in kernel-mode. Since KAISER maps the full address space in kernel-mode, the attacker can bypass the address-space isolation. Thus, KAISER does not mitigate Spectre attacks.

The implementation of the **translation-level recovery attack**, as described in Section 2.4, performed an unintended Spectre-BTB attack. Schwarzl et al. [90] showed that the virtual address stored in register `R14` caused the information leak. The Linux kernel v4.10 entry code used this register to load an address after an indirect branch. Gruss et al. [28] assumed that the `PREFETCHh` instruction caused the timing leak because Spectre attacks were not known at this time. The KAISER patch did not counter the prefetch side-channel attack. KAISER modified the kernel entry and exit code. By code changes, it slightly changed the registers used by the virtual address resolution that caused the leak [90].

6.3.2 Meltdown

Meltdown attacks, as Spectre attacks, use microarchitectural leaks of rolled-back transient execution. Instead of branch-prediction-based speculative execution, Meltdown uses faults with out-of-order execution to leak data. Modern CPUs can reorder and parallelize instruction execution of a sequential program to optimize the CPU load. When the CPU detects a fault within an operation, it reverts the state of ahead-of-time computed instructions, and then handles the fault. Even though the CPU discards results of instructions after the fault, it still leaks information through side effects. It does not revert CPU buffer and cache entries. There are a number of variants of Meltdown, as Figure 6.3 shows.

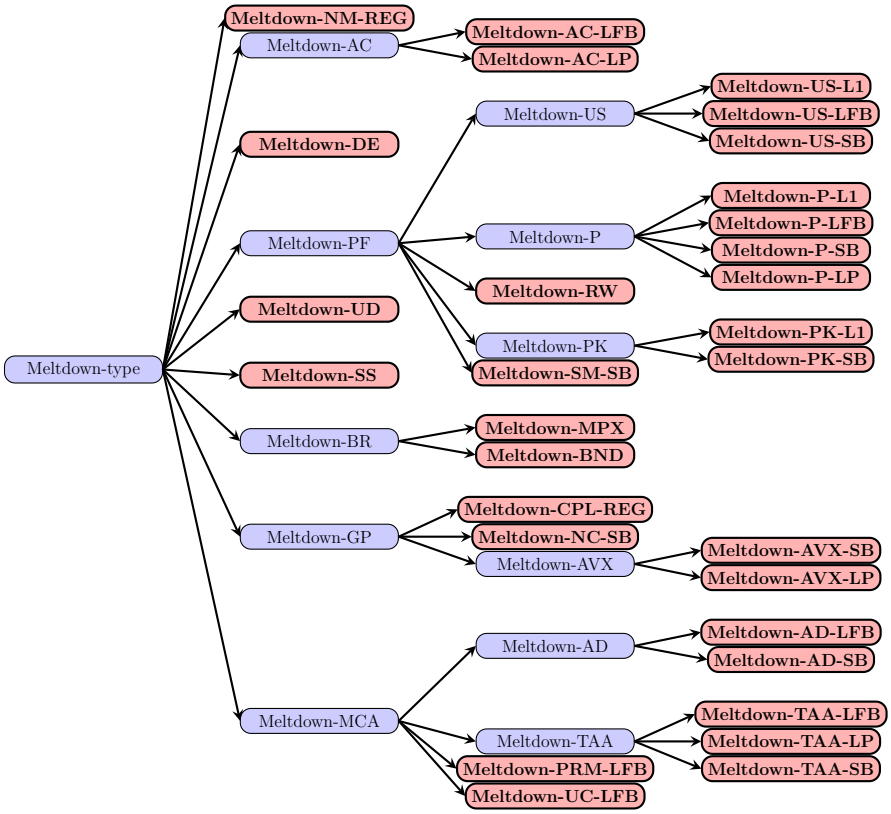

Figure 6.3: This figure visualizes the different variants of Meltdown attacks. The non-leaf nodes of the graph represent attack groups, and the leaf nodes are actual attacks [10, 81].

`Meltdown-US-L1` [73] (also known as CVE-2017-5754, Rogue Data Cache Load, and Variant 3 [42, 79]) is the first published Meltdown attack. It allows an attacker to read protected kernel memory from a user-mode thread. The Meltdown-US variants use a transient-execution leak in user mode to bypass the paging-structure privilege protection mechanism. When a user-mode thread accesses or executes a kernel address, the CPU resolves the virtual address. Since the corresponding TLB, paging-structure cache entry, or paging-structure entry user (or supervisor) flag is zero, the CPU triggers a page fault. Due to out-of-order-execution, the CPU out-of-order executes following instructions and reverts the results once it detects the access violation. Lipp et al. [73] found out that the CPU can leak a kernel value by instructions after the access violating instruction. If an attacker indirectly accesses a probe array based on protected kernel data, the CPU does not invalidate the L1 data cache entry when it reverts the instruction execution. The attacker can recover the protected kernel memory value through e.g., via a Flush+Reload attack.

Listing 6.2 shows a simplified example of the core part of Meltdown-US-L1. The attacking user thread first flushes the data caches of a probe array. It then accesses the secret kernel address. Directly afterward, the thread indirectly accesses the probe array by `secret * PAGE_SIZE`. The CPU transiently loads and the value, and updates the corresponding cache line entry. Then, it detects the access violation and throws an exception. There are several ways to deal with the fault. The attacker can handle the exception by forking a second process and using a shared probe array to recover the secret. The attacker can also set up a signal handler and recover the secret when the kernel attempts to kill the attacker's process. To suppress the exception entirely, the attacker can use an Intel TSX RTM region [73]. The attacker can also suppress the exception entirely by using a branch miss-prediction speculative execution (similar to Spectre). Once the attacker suppressed or handled the fault, the attacker probes (reloads) the probe array page-wise. From this step, the attacker can recover the secret by measuring the access time [73].

```
/* Dereference a kernel address, so the CPU triggers an
    exception. */
x = *(uint8_t*)ptr_secret;
/* The CPU accesses the probe array out-of-order, even
   though it should never reach the instruction. */
y = probe_array[x*4096];
```

Listing 6.2: A Meltdown-US-L1 code snippet, which is executed by a thread in user mode to read kernel data [73].

Meltdown-US-L1 needs the kernel memory to be mapped in user mode. Since KAISER removes most of the kernel-space mapping when a thread is in user mode, it mitigates the Meltdown-US-L1 attack [73].

6.4 Meltdown Mitigations

Since the Meltdown-US-L1 attack can readout protected kernel memory with up to 503 kB/s, Lipp et al. recommended to immediately deploy KAISER on every operating system kernel [73]. Linux kernel developers merged KPTI (aka KAISER) into the kernel v4.15-rc5, and back-ported KAISER for the long-term support kernels v4.9 [55] and v4.4 [54]. They also added support for 32-bit x86 kernels in Linux kernel v4.19 [72].

Windows, macOS, and iOS also implemented countermeasures against Melt-

down [6, 47]. These countermeasures implement the same idea and principle that we proposed with KAISER.

New Intel processors starting with the Intel Whiskey Lake architecture and AMD processors mitigate Meltdown-US-L1 by design and do not need a software mitigation [10, 45]. Still, KAISER adds additional protection against certain side-channel attacks.

Chapter 7

Conclusion

In this thesis we presented a novel security mechanism we designed and developed, namely KAISER. KAISER is a mechanism and proof-of-concept implementation to protect against attacks on KASLR. It unmaps the kernel virtual memory when a thread is in user mode, and flushes TLB and paging-structure caches whenever the kernel switches between user mode and kernel mode. We developed the KAISER implementation for Linux kernel `v4.10-rc6` for x86_64 CPUs. Whenever a thread enters or exits the kernel-mode by an interrupt, exception, or system call, the kernel switches the CR3 to a shadow paging-structure. The shadow mapping unmaps most of the Linux kernel, leaving the necessary entry code, exit code, and structures that the CPU needs, mapped in user mode. KAISER also stops the use of the global flag in kernel paging-structure tables, so the CPU invalidates all entries of the TLBs and the paging-structure caches upon context switches. When the thread switches back into the kernel mode, the entry handlers switch from the shadow mapping to the full kernel mapping again.

We evaluated the performance impact and memory footprint of KAISER and showed that KAISER only adds ≈1 MB memory and ≈0.28 % performance overhead on an Intel Skylake CPU. The security evaluation showed that our countermeasures mitigate double page-fault attacks, Intel TSX-based side channels, and translation-level recovery attacks, and, consequently, contributes to the leakage-resilience of KASLR implementations.

The idea of KAISER is to provide stronger isolation for kernel memory and we advised deployment as a precaution. Shortly after our initial publication of the KAISER idea and patch, Meltdown, a powerful transient execution attack, was discovered. Meltdown can directly read kernel memory if it is mapped in an address space shared between user and kernel, *i.e.*, the traditional address space

design in modern operating systems. Since KAISER removes the kernel memory mappings in user mode, KAISER turned out to be a software countermeasure against this new attack. This directly confirmed our initial recommendation to deploy KAISER as a precaution.

Kernel developers reworked the KAISER proof-of-concept patch. They renamed KAISER to KPTI and merged it into the Linux mainstream kernel version 4.15. They also back-ported it to other Linux long-term kernel versions and added support for x86 and Arm AArch64 processors. The overheads of KPTI in practice are between $-5\,\%$ and $800\,\%$, depending on processor and system activity. Microsoft Windows and Apple iOS and macOS implemented our idea from scratch in their operating systems. Thus, our idea is now implemented in all major operating systems and used by billions of users worldwide.

Bibliography

[1] 0xAX. Documentation/x86/kernel-stacks - kernel version 4.10-rc6, 08 2015. URL `https://github.com/torvalds/linux/blob/v4.10-rc6/Documentation/x86/kernel-stacks`.

[2] 0xAX. Linux Inside, 2020. URL `https://0xax.gitbooks.io/linux-insides/content/`.

[3] Onur Acıiçmez, Çetin Kaya Koç, and Jean-pierre Seifert. On the Power of Simple Branch Prediction Analysis. In *CCS*, 2007.

[4] Onur Acıiçmez, Jean-Pierre Seifert, and Çetin Kaya Koç. Predicting secret keys via branch prediction. In *CT-RSA*, 2007.

[5] Hans Peter Anvin. [tip:x86/espfix] x86-64, espfix: Don't leak bits 31: 16 of %esp returning to 16-bit stack, 04 2014. URL `https://lore.kernel.org/patchwork/patch/460889/`.

[6] Apple. About speculative execution vulnerabilities in ARM-based and Intel CPUs, 2018. URL `https://support.apple.com/en-us/HT208394`.

[7] ARM. *ARM Architecture Reference Manual*. ARM Limited, 2005.

[8] ARM. Cache Speculation Side-channels Version 2.5, 2020. URL `https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability`.

[9] Eric Bodden, Mathias Payer, and Elias Athanasopoulos, editors. *Engineering Secure Software and Systems*, volume 10379 of *Lecture Notes in Computer Science*. Springer International Publishing, 2017. ISBN 978-3-319-62104-3. doi: 10.1007/978-3-319-62105-0.

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In

USENIX Security Symposium, 2019. Extended classification tree and PoCs at https://transient.fail/.

[11] Jonathan Corbet. On vsyscalls and the vDSO, 06 2011. URL `https://lwn.net/Articles/446528/`.

[12] Jonathan Corbet. 3.14 Merge window part 1 [LWN.net], 2014. URL `https://lwn.net/Articles/581657/`.

[13] Jonathan Corbet. KAISER: hiding the kernel from user space, 2017. URL `https://lwn.net/Articles/738975/`.

[14] Jonathan Corbet. The current state of kernel page-table isolation, 12 2017. URL `https://lwn.net/Articles/741878/`.

[15] Jonathan Corbet. C library system-call wrappers, or the lack thereof, 2018. URL `https://lwn.net/Articles/771441/`.

[16] Intel Corporation. 5-Level Paging and 5-Level EPT White Paper Revision 1.1, 2017. URL `www.intel.com/design/literature.htm`.

[17] Will Deacon. [PATCH v2 00/18] arm64: Unmap the kernel whilst running in userspace (KAISER), 11 2017. URL `https://lkml.org/lkml/2017/11/30/594`.

[18] Izik Eidus and Hugh Dickins. Kernel Samepage Merging, 2009. URL `https://github.com/torvalds/linux/blob/v4.10-rc6/Documentation/vm/ksm.txt`.

[19] Daniel Eriksen. The Linux Boot Process, 2004. URL `https://bglug.ca/articles/linux_boot_process.pdf`.

[20] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *MICRO*, 2016.

[21] Anders Fogh. Micro architecture attacks on KASLR, 2016. URL `https://cyber.wtf/2016/10/25/micro-architecture-attacks-on-kasrl/`.

[22] Anders Fogh. Negative Result: Reading Kernel Memory From User Mode, 2017. URL `https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/`.

[23] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. LAZARUS: Practical Side-Channel Resilient Kernel-Space Randomization. In *RAID*, 2017.

[24] Matt Godbolt. The BTB in contemporary Intel chips, 02 2016. URL https://xania.org/201602/bpu-part-three.

[25] Mariano Graziano, Davide Balzarotti Eurecom, and Alain Zidouemba. ROPMEMU: A Framework for the Analysis of Complex Code-Reuse Attacks. In *CCS*, pages 47–58, 2016.

[26] Brendan Gregg. KPTI/KAISER Meltdown Initial Performance Regressions, 2018. URL http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html.

[27] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*, 2015.

[28] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS*, 2016.

[29] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*, 2016.

[30] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In *ESSoS*, 2017.

[31] Daniel Gruss, Dave Hansen, and Brendan Gregg. Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer. *USENIX ;login*, 2018.

[32] Dave Hansen. [PATCH 00/23] KAISER: unmap most of the kernel from userspace page tables, 10 2017. URL https://lkml.org/lkml/2017/10/31/884.

[33] Dave Hansen. [PATCH 00/30] [v3] KAISER: unmap most of the kernel from userspace page tables, 11 2017. URL https://lkml.org/lkml/2017/11/10/433.

[34] Seokhie Hong. *Side Channel Attacks*. MDPI, 6 2019. ISBN 978-3-03921-001-5. doi: 10.3390/books978-3-03921-001-5. URL http://www.mdpi.com/books/pdfview/book/1339.

[35] Andy Honig and Kees Cook. x86, kaslr: randomize module base load address - Patchwork, 02 2014. URL https://lore.kernel.org/patchwork/patch/444214/.

[36] Jann Horn. Reading privileged memory with a side-channel, 2018. URL https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html.

[37] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P*, 2013.

[38] IBM. Potential Impact on Processors in the POWER Family, 2018. URL https://www.ibm.com/blogs/psirt/potential-impact-processors-power-family/.

[39] Advanced Micro Devices Inc. AMD64 Architecture Programmer's Manual, 2020.

[40] Advanced Micro Devices Inc. AMD Product Security, 2020. URL https://www.amd.com/en/corporate/product-security.

[41] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide, 2016.

[42] Intel. Rogue Data Cache Load / CVE-2017-5754 / INTEL-SA-00088, 2018. URL https://software.intel.com/security-software-guidance/software-guidance/rogue-data-cache-load.

[43] Intel. Speculative Execution Side Channel Mitigations, 2018. Revision 3.0.

[44] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2019.

[45] Intel. Engineering New Protections Into Hardware, 04 2019. URL https://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html.

[46] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *CCS*, 2016.

[47] Ken Johnson. KVA Shadow: Mitigating Meltdown on Windows, 03 2018. URL https://blogs.technet.microsoft.com/srd/2018/03/23/kva-shadow-mitigating-meltdown-on-windows/.

[48] Timothy M Jones. Inside the Linux boot process, 2006. URL https://developer.ibm.com/technologies/linux/articles/l-linuxboot/.

[49] Matthew Kerner and Neil Padgett. A History of Modern 64-bit Computing, 2007. URL http://courses.cs.washington.edu/courses/csep590/06au/projects/history-64-bit.pdf.

[50] Michael Kerrisk. *The Linux Programming Interface*. No Starch Press, 2010. ISBN 9781593272203. URL http://man7.org/.

[51] Andi Kleen. Documentation/x86/x86_64/mm.txt - kernel version 4.10-rc6, 2004. URL https://github.com/torvalds/linux/blob/v4.10-rc6/Documentation/x86/x86_64/mm.txt.

[52] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[53] Paul C. Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, 1996.

[54] Greg Kroah-Hartman. Linux 4.4.110, 1 2018. URL https://lkml.org/lkml/2018/1/5/332.

[55] Greg Kroah-Hartman. Linux 4.9.75, 01 2018. URL https://lkml.org/lkml/2018/1/5/335.

[56] Christoph Lameter and Pranith Kumar. Documentation/this_cpu_ops.txt - kernel version 4.10-rc6, 9 2014. URL https://github.com/torvalds/linux/blob/v4.10-rc6/Documentation/this_cpu_ops.txt.

[57] ARM Limited. Application Note Migrating from IA-32 to ARM, 2011. URL http://infocenter.arm.com/help/topic/com.arm.doc.dai0274b/DAI0274B_migrating_from_ia32_to_arm.pdf.

[58] Arm Limited. ARM: Media fact sheet (Sept. 1, 2016), 2017. URL https://www.arm.com/-/media/arm-com/news/ARM-media-fact-sheet-2016.pdf.

[59] Linux. Documentation/x86/entry_64.txt - kernel version 4.10-rc6, 07 2015. URL https://github.com/torvalds/linux/blob/v4.10-rc6/Documentation/x86/entry_64.txt.

[60] Linux. arch/x86/include/asm/irq_vectors.h - kernel version 4.10-rc6, 07 2015. URL https://github.com/torvalds/linux/blob/v4.10-rc6/arch/x86/include/asm/irq_vectors.h.

[61] Linux. linux/arch/x86/Kconfig - kernel version 4.10-rc6, 2016. URL https://github.com/torvalds/linux/blob/v4.10-rc6/arch/x86/Kconfig.

[62] Linux. Documentation/x86/boot.txt - kernel version 4.10-rc6, 10

2016. URL https://github.com/torvalds/linux/blob/v4.10-rc6/Documentation/x86/boot.txt.

[63] Linux. init/main.c - kernel version 4.10-rc6, 12 2016. URL https://github.com/torvalds/linux/blob/v4.10-rc6/init/main.c.

[64] Linux. Documentation/scheduler/sched-design-CFS.txt - kernel version 4.10-rc6, 8 2016. URL https://github.com/torvalds/linux/blob/v4.10-rc6/Documentation/scheduler/sched-design-CFS.txt.

[65] Linux. arch/x86/kernel/setup_percpu.c - kernel version 4.10-rc6, 11 2016. URL https://github.com/torvalds/linux/blob/v4.10-rc6/arch/x86/kernel/setup_percpu.c.

[66] Linux. arch/x86/entry/entry_64.S - kernel version 4.10-rc6, 01 2017. URL https://github.com/torvalds/linux/blob/v4.10-rc6/arch/x86/entry/entry_64.S.

[67] Linux. Linux Kernel User Documentation Release - kernel version 4.13.0-rc4, 2017. URL http://kernel.org/.

[68] Linux. syscalls(2) - Linux manual page, 2020. URL http://man7.org/linux/man-pages/man2/syscalls.2.html.

[69] Linux. modify_ldt(2) - Linux manual page, 2 2020. URL https://man7.org/linux/man-pages//man2/modify_ldt.2.html.

[70] Linux Kernel Newbies. Linux_4.14, 11 2017. URL https://kernelnewbies.org/Linux_4.14.

[71] Linux Kernel Newbies. Linux_4.15, 1 2018. URL https://kernelnewbies.org/Linux_4.15.

[72] Linux Kernel Newbies. Linux_4.19, 11 2018. URL https://kernelnewbies.org/Linux_4.19.

[73] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

[74] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*, 2015.

[75] Andrew Lutomirski. /arch/x86/entry/vdso/vdso32/system_call.S - kernel version 4.10-rc6, 6 2016. URL https://github.com/torvalds/linux/blob/v4.10-rc6/arch/x86/entry/vdso/vdso32/system_call.S.

[76] Hector Marco-Gisbert and Ismael Ripoll Ripoll. Address Space Layout Randomization Next Generation. *Applied Sciences*, 9, 7 2019. URL https://www.mdpi.com/2076-3417/9/14/2928.

[77] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *ACM SIGARCH Computer Architecture News*, 2012.

[78] Milena Milenkovic, Aleksandar Milenkovic, and Jeffrey Kulick. Demystifying intel branch predictors. In *Workshop on Duplicating, Deconstructing and Debunking*, 05 2002.

[79] MITRE. CVE-2017-5754, 2017. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5754.

[80] Ingo Molnar. 4G/4G split on x86, 64 GB RAM (and more) support, 7 2003. URL https://lkml.org/lkml/2003/7/8/246.

[81] Graz University of Technology. Transient Execution Attacks, 2020. URL https://transient.fail/.

[82] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*, 2006.

[83] PaX Team. Address space layout randomization (ASLR), 2003.

[84] Colin Percival. Cache missing for fun and profit. In *BSDCan*, 2005.

[85] Princeton University. The PARSEC Benchmark Suite, 2009. URL https://parsec.cs.princeton.edu/parsec3-doc.htm.

[86] Mike Rapoport. Memory: the flat, the discontiguous, and the sparse, 27 2019. URL https://lwn.net/Articles/789304/.

[87] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. In *ACM Transactions on Information and System Security*, volume 15, pages 1–34, 3 2012.

[88] rppt. Documentation/vm/memory-model.rst - kernel version 5.2-rc1, 4 2019. URL https://github.com/torvalds/linux/blob/v5.2-rc1/Documentation/vm/memory-model.rst.

[89] Rusling, David. The Linux Kernel, 1999. URL https://tldp.org/LDP/tlk/tlk.html.

[90] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel

Gruss. Speculative Dereferencing of Registers: Reviving Foreshadow. *arXiv:2008.02307v1*, 8 2020.

[91] Taku Shimosawa. Linux Kernel Booting Process (1) - For NLKB, 2014. URL `https://www.slideshare.net/shimosawa/linux-kernel-booting-process-1-for-nlkb`.

[92] James E. Smith and Gurindar S. Sohi. The Microarchitecture of Superscalar Processors. *IEEE*, 83:1609–1624, 1995.

[93] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. Systematic classification of side-channel attacks: a case study for mobile devices. *IEEE*, 20(1):465–488, 2017.

[94] Andrew Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson, 4 edition, 2014. ISBN 978-0-13-359162-0.

[95] The PostgreSQL Global Development Group. pgbench, 02 2016. URL `https://www.postgresql.org/docs/9.6/static/pgbench.html`.

[96] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967.

[97] Linus Torvalds. active_mm, 1999. URL `https://marc.info/?l=linux-kernel&m=93337278602211&w=2`.

[98] IDC Corporate USA. IDC Worldwide Quarterly Server Tracker, 2017. URL `https://www.idc.com/getdoc.jsp?containerId=prUS42335617`.

[99] Girish Venkatasubramanian, Renato J. Figueiredo, Ramesh Illikkal, and Donald Newell. TMT - A TLB tag management framework for virtualized platforms. In *HPCA*, pages 153–160, 2009.

[100] Xen Community. Introduction to Xen 3.x, 2016. URL `https://wiki.xenproject.org/wiki/Introduction_to_Xen_3.x`.

[101] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.

[102] Younis A. Younis, Kashif Kifayat, Qi Shi, and Bob Askwith. A New Prime and Probe Cache Side-Channel Attack for Cloud Computing. In *2015 IEEE International Conference on Computer and Information Technology (CIT)*, pages 1718–1724. IEEE, 10 2015. URL `http://ieeexplore.ieee.org/document/7363305/`.

Appendix A

Acronyms

APIC Advanced Programmable Interrupt Controller.

ARM Advanced RISC Machine.

ASLR Address Space Layout Randomization.

BIOS Basic Input/Output System.

BPU Branch Prediction Unit.

BTB Branch Target Buffer.

CISC Complex Instruction Set Computing.

COW Copy-On-Write.

CPL Current Privilege Level.

CPU Central Processing Unit.

CR3 Control Register 3.

CR4 Control Register 4.

CS Code Segment.

DPL Descriptor Privilege Level.

DrK De-Randomize Kernel address space.

DS Data Segment.

EFI Extensible Firmware Interface.

EFLAGS FLAGS Register E.

ES Extra Segment.

FS General Purpose Segment F.

GDT Global Descriptor Table.

GDTR Global Descriptor Table Register.

GRUB Grand Unified Bootloader.

GS General Purpose Segment G.

HDD Hard Disk Drive.

HLE Hardware Lock Elision.

IBPB Indirect Branch Predictor Barrier.

IDT Interrupt Descriptor Table.

IO Input/Output.

IRQ Interrupt Request.

ISA Instruction Set Architecture.

ISR Interrupt Service Routine.

IST Interrupt Stack Table.

KAISER Kernel Address Isolation to have Side-channels Efficiently Removed.

KASAN KernelAddressSanitizer.

KASLR Kernel Address Space Layout Randomization.

KPTI Kernel Page Table Isolation.

KS Kernel Space.

KSM Kernel Samepage Merging.

KVM Kernel Virtual Memory.

LDT Local Descriptor Table.

LLC Last Level Cache.

MCE Machine Check Exception.

MMU Memory Management Unit.

MSRS Model-Specific Registers.

NMI Non-Maskable Interrupt.

PCID Process-Context Identifier.

PD Page Directory.

PDPT Page Directory Pointer.

PFN Page-Frame Number.

PGD Page Global Directory.

PHT Pattern History Table.

PMD Page Middle Directory.

PML4 Page Map Level 4.

POSIX Portable Operating System Interface.

PPN Physical Page Number.

PT Page Table.

PUD Page Upper Directory.

RAM Random Access Memory.

RIP Register Instruction Pointer.

RISC Reduced Instruction Set Computing.

ROP Return-oriented Programming.

RSP Register Stack Pointer.

RTM Restricted Transactional Memory.

SMAP Supervisor Mode Access Protection.

SMEP Supervisor Mode Execution Protection.

SMP Symmetric Multiprocessing.

SS Stack Segment.

SSD Solid State Drive.

TI Table Indicator.

TLB Translation Lookaside Buffer.

TSD Time Stamp Disable.

TSS Task State Segment.

TSX Transactional Synchronization Extensions.

TTBR Translation Table Base Register.

UEFI Unified Extensible Firmware Interface.

VM Virtual Memory.

VTSC Virtual Time Stamp Counter.

W⊕X Write XOR Execute.