KASLR is Dead: Long Live KASLR

Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard

Graz University of Technology, Austria

Abstract. Modern operating system kernels employ address space layout randomization (ASLR) to prevent control-flow hijacking attacks and code-injection attacks. While kernel security relies fundamentally on preventing access to address information, recent attacks have shown that the hardware directly leaks this information. Strictly splitting kernel space and user space has recently been proposed as a theoretical concept to close these side channels. However, this is not trivially possible due to architectural restrictions of the x86 platform.
In this paper we present *KAISER*, a system that overcomes limitations of x86 and provides practical kernel address isolation. We implemented our proof-of-concept on top of the Linux kernel, closing all hardware side channels on kernel address information. *KAISER* enforces a strict kernel and user space isolation such that the hardware does not hold any information about kernel addresses while running in user mode. We show that *KAISER* protects against double page fault attacks, prefetch side-channel attacks, and TSX-based side-channel attacks. Finally, we demonstrate that *KAISER* has a runtime overhead of only 0.28%.

1 Introduction

Like user programs, kernel code contains software bugs which can be exploited to undermine the system security. Modern operating systems use hardware features to make the exploitation of kernel bugs more difficult. These protection mechanisms include making code non-writable and data non-executable. Moreover, accesses from kernel space to user space require additional indirection and cannot be performed through user space pointers directly anymore (SMAP/SMEP). However, kernel bugs can be exploited within the kernel boundaries. To make these attacks harder, address space layout randomization (ASLR) can be used to make some kernel addresses or even all kernel addresses unpredictable for an attacker. Consequently, powerful attacks relying on the knowledge of virtual addresses, such as return-oriented-programming (ROP) attacks, become infeasible [14,17,19]. It is crucial for kernel ASLR to withhold any address information from user space programs. In order to eliminate address information leakage, the virtual-to-physical address information has been made unavailable to user programs [13].

Knowledge of virtual or physical address information can be exploited to bypass KASLR [7,22], bypass SMEP and SMAP [11], perform side-channel attacks [6,15,18], Rowhammer attacks [5,12,20], and to attack system memory encryption [2]. To prevent attacks, system interfaces leaking the virtual-to-physical mapping have recently been fixed [13]. However, hardware side channels might not easily be fixed without changing the hardware. Specifically side-channel attacks targeting the page translation caches provide information about virtual and physical addresses to the user space. Hund et al. [7] described an attack exploiting double page faults, Gruss et al. [6] described an attack exploiting software prefetch instructions,[1] and Jang et al. [10] described an attack exploiting Intel TSX (hardware transactional memory). These attacks show that current KASLR implementations have fatal flaws, subsequently KASLR has been proclaimed dead by many researchers [3, 6, 10].

Gruss et al. [6] and Jang et al. [10] proposed to unmap the kernel address space in the user space and vice versa. However, this is non-trivial on modern x86 hardware. First, modifying page table structures on context switches is not possible due to the highly parallelized nature of today's multi-core systems, e.g., simply unmapping the kernel would inhibit parallel execution of multiple system calls. Second, x86 requires several locations to be valid for both user space and kernel space during context switches, which are hard to identify in large operating systems. Third, switching or modifying address spaces incurs translation lookaside buffer (TLB) flushes [8]. Jang et al. [10] suspected that switching address spaces may have a severe performance impact, making it impractical.

In this paper, we present *KAISER*, a highly-efficient practical system for kernel address isolation, implemented on top of a regular Ubuntu Linux. *KAISER* uses a shadow address space paging structure to separate kernel space and user space. The lower half of the shadow address space is synchronized between both paging structures. Thus, multiple threads work in parallel on the two address spaces if they are in user space or kernel space respectively. *KAISER* eliminates the usage of global bits in order to avoid explicit TLB flushes upon context switches. Furthermore, it exploits optimizations in current hardware that allow switching address spaces without performing a full TLB flush. Hence, the performance impact of *KAISER* is only 0.28%.

KAISER reduces the number of overlapping pages between user and kernel address space to the absolute minimum required to run on modern x86 systems. We evaluate all microarchitectural side-channel attacks on kernel address information that are applicable to recent Intel architectures. We show that *KAISER* successfully eliminates the leakage in all cases.

Contributions. The contributions of this work are:

1. *KAISER* is the first practical system for kernel address isolation. It introduces shadow address spaces to utilize modern CPU features efficiently avoiding frequent TLB flushes. We show how all challenges to make kernel address isolation practical can be overcome.

[1] The list of authors for "Prefetch Side-Channel Attacks" by Gruss et al. [6] and this paper overlaps.

2. Our open-source proof-of-concept implementation in the Linux kernel shows that *KAISER* can easily be deployed on commodity systems, *i.e.*, a full-fledged Ubuntu Linux system.[2]
3. After KASLR has already been considered dead by many researchers, *KAISER* fully restores the former efficacy of KASLR with a runtime overhead of only 0.28%.

Outline. The remainder of the paper is organized as follows. In Section 2, we provide background on kernel protection mechanisms and side-channel attacks. In Section 3, we describe the design and implementation of *KAISER*. In Section 4, we evaluate the efficacy of *KAISER* and its performance impact. In Section 5, we discuss future work. We conclude in Section 6.

2 Background

2.1 Virtual Address Space

Virtual addressing is the foundation of memory isolation between different processes as well as processes and the kernel. Virtual addresses are translated to physical addresses through a multi-level translation table stored in physical memory. A CPU register holds the physical address of the active top-level translation table. Upon a context switch, the register is updated to the physical address of the top-level translation table of the next process. Consequently, processes cannot access all physical memory but only the memory that is mapped to virtual addresses. Furthermore, the translation tables entries define properties of the corresponding virtual memory region, e.g., read-only, user-accessible, non-executable.

On modern Intel x86-64 processors, the top-level translation table is the page map level 4 (PML4). Its physical address is stored in the `CR3` register of the CPU. The PML4 divides the 48-bit virtual address space into 512 PML4 entries, each covering a memory region of 512 GB. Each subsequent level sub-divides one block of the upper layer into 512 smaller regions until 4 kB pages are mapped using page tables (PTs) on the last level. The CPU has multiple levels of caches for address translation table entries, the so-called TLBs. They speed up address translation and privilege checks. The kernel address space is typically a defined region in the virtual address space, e.g., the upper half of the address space.

Similar translation tables exist on modern ARM (Cortex-A) processors too, with small differences in size and property bits. One significant difference to x86-64 is that ARM CPUs have two registers to store physical addresses of translation tables (TTBR0 and TTBR1). Typically, one is used to map the user address space (lower half) whereas the other is used to map the kernel address space (upper half). Gruss et al. [6] speculated that this might be one of the reasons why the attack does not work on ARM processors. As x86-64 has only

[2] We are preparing a submission of our patches into the Linux kernel upstream. The source code and the Debian package compatible with Ubuntu 16.10 can be found at `https://github.com/IAIK/KAISER`.

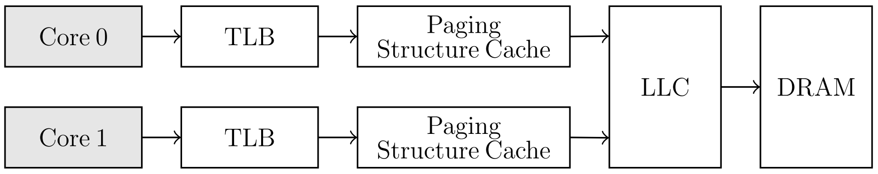


Fig. 1: Address translation caches are used to speed up address translation table lookups.

one translation-table register (`CR3`), it is used for both user and kernel address space. Consequently, to perform privilege checks upon a memory access, the actual page translation tables have to be checked.

Control-Flow Attacks. Modern Intel processors protect against code injection attacks through non-executable bits. Furthermore, code execution and data accesses on user space memory are prevented in kernel mode by the CPU features supervisor-mode access prevention (SMAP) and supervisor-mode execution prevention (SMEP). However, it is still possible to exploit bugs by redirecting the code execution to existing code. Solar Designer [23] showed that a non-executable stack in user programs can be circumvented by jumping to existing functions within libc. Kemerlis et al. [11] presented the *ret2dir* attack which redirects a hijacked control flow in the kernel to arbitrary locations using the kernel physical direct mapping. Return-oriented programming (ROP) [21] is a generalization of such attacks. In ROP attacks, multiple code fragments—so-called gadgets—are chained together to build an exploit. Gadgets are not entire functions, but typically consist of one or more useful instructions followed by a return instruction.

To mitigate control-flow-hijacking attacks, modern operating systems randomize the virtual address space. Address space layout randomization (ASLR) ensures that every process has a new randomized virtual address space, preventing an attacker from knowing or guessing addresses. Similarly, the kernel has a randomized virtual address space every time it is booted. As Kernel ASLR makes addresses unpredictable, it protects against ROP attacks.

2.2 CPU Caches

Caches are small memory buffers inside the CPU, storing frequently used data. Modern Intel CPUs have multiple levels of set-associative caches. The last-level cache (LLC) is shared among all cores. Executing code or accessing data on one core has immediate consequences for all other cores.

Address translation tables are stored in physical memory. They are cached in regular data caches [8] but also in special caches such as the translation lookaside buffers. Figure 1 illustrates how the address translation caches are used for address resolution.

2.3 Microarchitectural Attacks on Kernel Address Information

Until recently, Linux provided information on virtual and physical addresses to any unprivileged user program through operating system interfaces. As this information facilitates mounting microarchitectural attacks, the interfaces are now restricted [13]. However, due to the way the processor works, side channels through address translation caches [4, 6, 7, 10] and the branch-target buffer [3] leak parts of this information.

Address Translation Caches. Hund et al. [7] described a double page fault attack, where an unprivileged attacker tries to access an inaccessible kernel memory location, triggering a page fault. After the page fault interrupt is handled by the operating system, the control is handed back to an error handler in the user program. The attacker measures the execution time of the page fault interrupt. If the memory location is valid, regardless of whether it is accessible or not, address translation table entries are copied into the corresponding address translation caches. The attacker then tries to access the same inaccessible memory location again. If the memory location is valid, the address translation is already cached and the page fault interrupt will take less time. Thus, the attacker learns whether a memory location is valid or not, even if it is not accessible from the user space.

Jang et al. [10] exploited the same effect in combination with Intel TSX. Intel TSX is an extension to the x86 instruction set providing a hardware transactional memory implementation via so-called TSX transactions. If a page fault occurs within a TSX transaction, the transaction is aborted without any operating system interaction. Thus, the entire page fault handling of the operation system is skipped, and the timing differences are significantly less noisy. In this attack, the attacker again learns whether a memory location is valid, even if it is not accessible from the user space.

Gruss et al. [6] exploited software prefetch instructions to trigger address translation. The execution time of the prefetch instruction depends on which address translation caches hold the right translation entries. Thus, in addition to learning whether an inaccessible address is valid or not, an attacker learns its corresponding page size as well. Furthermore, software prefetches can succeed even on inaccessible memory. Linux has a kernel physical direct map, providing direct access to all physical memory. If the attacker prefetches an inaccessible address in this kernel physical direct map corresponding to a user-accessible address, it will also be cached when accessed through the user address. Thus, the attacker can retrieve the exact physical address for any virtual address.

All three attacks have in common that they exploit that the kernel address space is mapped in user space as well, and that accesses are only prevented through the permission bits in the address translation tables. Thus, they use the same entries in the paging structure caches. On ARM architectures, the user and kernel addresses are already distinguished based on registers, and thus no cache access and no timing difference occurs. Gruss et al. [6] and Jang et al. [10] proposed to unmap the entire kernel space to emulate the same behavior as on the ARM architecture.

Branch-Target Buffer. Evtyushkin et al. [3] presented an attack on the branch-target buffer (BTB) to recover the lowest 30 bits of a randomized kernel address. The BTB is indexed based on the lowest 30 bits of the virtual address. Similar as in a regular cache attack, the adversary occupies parts of the BTB by executing a sequence of branch instructions. If the kernel uses virtual addresses with the same value for the lowest 30 bits as the attacker, the sequence of branch instructions requires more time. Through targeted execution of system calls, the adversary can obtain information about virtual addresses of code that is executed during a system call. Consequently, the BTB attack defeats KASLR.

We consider the BTB attack out of scope for our countermeasure (*KAISER*), which we present in the next section, for two reasons. First, Evtyushkin et al. [3] proposed to use virtual address bits > 30 to randomize memory locations for KASLR as a zero-overhead countermeasure against their BTB attack. Indeed, an adaption of the corresponding range definitions in modern operating system kernels would effectively mitigate the attack. Second, the BTB attack relies on a profound knowledge of the behavior of the BTB. The BTB attack currently does not work on recent architectures like Intel Skylake, as the BTB has not been reverse-engineered yet. Consequently, we also were not able to reproduce the attack in our test environment (Intel Skylake i7-6700K).

3 Design and Implementation of *KAISER*

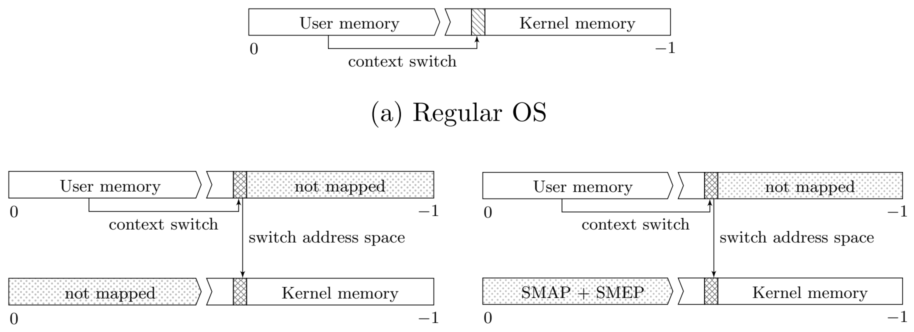
In this section, we describe the design and implementation of *KAISER*[3]. We discuss the challenges of implementing kernel address isolation. We show how shadow address space paging structures can be used to separate kernel space and user space. We describe how modern CPU features and optimizations can be used to reduce the amount of regular TLB flushes to a minimum. Finally, to show the feasibility of the approach, we implemented *KAISER* on top of the latest Ubuntu Linux kernel.

3.1 Challenges of Kernel Address Isolation

As recommended by Intel [8], today's operating systems map the kernel into the address space of every user process. Kernel pages are protected from unwanted access by user space applications using different access permissions, set in the page table entries (PTE). Thus, the address space is shared between the kernel and the user and only the privilege level is escalated to execute system calls and interrupt routines.

The idea of *Stronger Kernel Isolation* proposed by Gruss et al. [6] (cf. Figure 2) is to unmap kernel pages while the user process is in user space and switch to a separated kernel address space when entering the kernel. Consequently, user pages are not mapped in kernel space and only a minimal numbers of pages is mapped both in user space and kernel space. While this would prevent

[3] Kernel Address Isolation to have Side channels Efficiently Removed.



(a) Regular OS

(b) Stronger kernel isolation

(c) KAISER

Fig. 2: (a) The kernel is mapped into the address space of every user process. (b) Theoretical concept of stronger kernel isolation. It splits the address spaces and only interrupt handling code is mapped in both address spaces. (c) For compatibility with x86 Linux, KAISER relies on SMAP to prevent invalid user memory references and SMEP to prevent execution of user code in kernel mode.

all microarchitectural attacks on kernel address space information on recent systems [6, 7, 10], it is not possible to implement *Stronger Kernel Isolation* without rewriting large parts of today's kernels. There is no previous work investigating the requirements real hardware poses to implement kernel address isolation in practice. We identified the following three challenges that make kernel address isolation non-trivial to implement.

Challenge 1. Threads cannot use the same page table structures in user space and kernel space without a huge synchronization overhead. The reason for this is the highly parallelized nature of today's systems. If a thread modifies page table structures upon a context switch, it influences all concurrent threads of the same process. Furthermore, the mapping changes for all threads, even if they are currently in the user space.

Challenge 2. Current x86 processors require several locations to be valid for both user space and kernel space during context switches. These locations are hard to identify in large operating system kernels due to implicit assumptions about the omnipresence of the entire kernel address space. Furthermore, segmented memory accesses like core-local storage are required during context switches. Thus, it must be possible to locate and restore the segmented areas without re-mapping the unmapped parts of the kernel space. Especially, unmapping the user space in the Linux kernel space, as proposed by Gruss et al. [6], would require rewriting large parts of the Linux kernel.

Challenge 3. Switching the address space incurs an implicit full TLB flush and modifying the address space causes a partial TLB flush [8]. As current operating systems are highly optimized to reduce the amount of implicit TLB flushes, a countermeasure would need to explicitly flush the TLB upon every context switch. Jang et al. [10] suspected that this may have a severe performance impact.

3.2 Practical Kernel Address Isolation

In this section we show how *KAISER* overcomes these challenges and thus fully revives KASLR.

Shadow Address Spaces. To solve challenge 1, we introduce the idea of *shadow address spaces* to provide kernel address isolation. Figure 3 illustrates the principle of the shadow address space technique. Every process has two address spaces. One address space which has the user space mapped but not the kernel (*i.e.*, the *shadow address space*), and a second address space which has the kernel mapped but the user space protected with SMAP and SMEP.

The switch between the user address space and the kernel address space now requires updating the `CR3` register with the value of the corresponding PML4. Upon a context switch, the `CR3` register initially remains at the old value, mapping the user address space. At this point *KAISER* can only perform a very limited amount of computations, operating on a minimal set of registers and accessing only parts of the kernel that are mapped both in kernel and user space. As interrupts can be triggered from both user and kernel space, interrupt sources can be both environments and it is not generally possible to determine the interrupt source within the limited amount of computations we can perform at this point. Consequently, switching the `CR3` register must be a short static computation oblivious to the interrupt source.

With shadow address spaces we provide a solution to this problem. Shadow address spaces are required to have a globally fixed power-of-two offset between the kernel PML4 and the shadow PML4. This allows switching to the kernel PML4 or the shadow PML4 respectively, regardless of the interrupt source. For instance, setting the corresponding address bit to zero switches to the kernel PML4 and setting it to one switches to the shadow PML4. The easiest offset to implement is to use bit 12 of the physical address. That is, the PML4 for the kernel space and shadow PML4 are allocated as an 8 kB-aligned physical memory block. The shadow PML4 is always located at the offset +4 kB. With this trick, we do not need to perform any memory lookups and only need a single scratch register to switch address spaces.

The memory overhead introduced through shadow address spaces is very small. We have an overhead of 8 kB of physical memory per user thread for kernel page directorys (PDs) and PTs and 12 kB of physical memory per user process for the shadow PML4. The 12 kB are due to a restriction in the Linux kernel that only allows to allocate blocks containing 2^n pages. Additionally, *KAISER* has a system-wide total overhead of 1 MB to allocate 256 global kernel page directory pointer tables (PDPTs) that are mapped in the kernel region of the shadow address spaces.

Minimizing the Kernel Address Space Mapping. To solve challenge 2, we identified the memory regions that need to be mapped for both user space

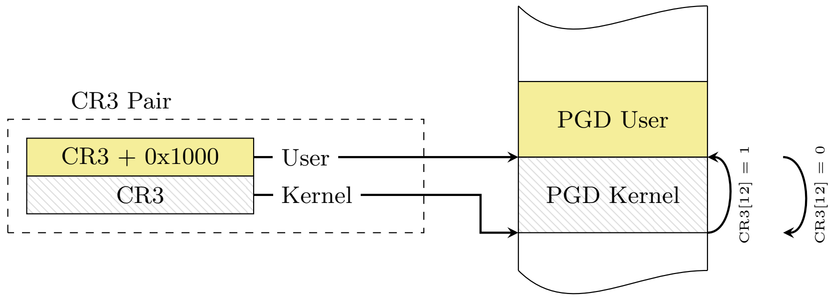


Fig. 3: Shadow address space: PML4 of user address space and kernel address space are placed next to each other in physical memory. This allows to switch between both mappings by applying a bit mask to the `CR3` register.

and kernel space, *i.e.*, the absolute minimum number of pages to be compatible with x86 and its features used in the Linux kernel. While previous work [6] suggested that only a negligible portion of the interrupt dispatcher code needs to be mapped in both address spaces, in practice more locations are required.

As x86 and Linux are built around using interrupts for context switches, it is necessary to map the interrupt descriptor table (IDT), as well as the interrupt entry and exit `.text` section. To enable multi-threaded applications to run on different cores, it is necessary to identify per-CPU memory regions and map them into the shadow address space. *KAISER* maps the entire per-CPU section including the interrupt request (IRQ) stack and vector, the global descriptor table (GDT), and the task state segment (TSS). Furthermore, while switching to privileged mode, the CPU implicitly pushes some registers onto the current kernel stack. This can be one of the per-CPU stacks that we already mapped or a thread stack. Consequently, thread stacks need to be mapped too.

We found that the idea to unmap the user space entirely in kernel space is not practical. The design of modern operating system kernels is based upon the capability of accessing user space addresses from kernel mode. Furthermore, SMEP protects against executing user space code in kernel mode. Any memory location that is user-accessible cannot be executed by the kernel. SMAP protects against invalid user memory references in kernel mode. Consequently, the effective user memory mapping is non-executable and not directly accessible in kernel mode.

Efficient and Secure TLB Management. The Linux kernel generally tries to minimize the number of implicit TLB flushes. For instance when switching between kernel and user mode, the `CR3` register is not updated. Furthermore, the Linux kernel uses PTE global bits to preserve mappings that exist in every process to improve the performance of context switches. The global bit of a PTE marks pages to be excluded from implicit TLB flushes. Thus, they reduce the impact of implicit TLB flushes when modifying the `CR3` register.

To solve challenge 3, we investigate the effects of these global bits. We found that it is necessary to either perform an explicit full TLB flush, or disable the global bits to eliminate the leakage completely. Surprisingly, we found the performance impact of disabling global bits to be entirely negligible.

Disabling global bits alone does not eliminate any leakage, but it is a necessary building block. The main side-channel defense in *KAISER* is based on the separate shadow address spaces we described above. As the two address spaces have different `CR3` register values, *KAISER* requires a `CR3` update upon every context switch. The defined behavior of current Intel x86 processors is to perform implicit TLB flushes upon every `CR3` update. Venkatasubramanian et al. [25] described that beyond this architecturally defined behavior, the CPU may implement further optimizations as long as the observed effect does not change. They discussed an optimized implementation which tags the TLB entries with the `CR3` register to avoid frequent TLB flushes due to switches between processes or between user mode and kernel mode. As we show in the following section, our evaluation suggests that current Intel x86 processors have such optimizations already implemented. *KAISER* benefits from these optimizations implicitly and consequently, its TLB management is efficient.

4 Evaluation

We evaluate and discuss the efficacy and performance of *KAISER* on a desktop computer with an Intel Core i7-6700K Skylake CPU and 16GB RAM. To evaluate the effectiveness of *KAISER*, we perform all three microarchitectural attacks applicable to Skylake CPUs (cf. Section 2). We perform each attack with and without *KAISER* enabled and show that *KAISER* can mitigate all of them. For the performance evaluation, we compare various benchmark suites with and without *KAISER* and observe a negligible performance overhead of only 0.08 % to 0.68 %.

4.1 Evaluation of Microarchitectural Attacks

Double Page Fault Attack. As described in Section 2, the double page fault attack by Hund et al. [7] exploits the fact that the page translation caches store information to valid kernel addresses, resulting in timing differences. As *KAISER* does not map the kernel address space, kernel addresses are never valid in user space and thus, are never cached in user mode. Figure 4 shows the average execution time of the second page fault. For the default kernel, the execution time of the second page fault is 12 282 cycles for a mapped address and 12 307 cycles for an unmapped address. When running the kernel with *KAISER*, the access time is 14 621 in both cases. Thus, the leakage is successfully eliminated.

Note that the observed overhead for the page fault execution does not reflect the actual performance penalty of *KAISER*. The page faults triggered for this attack are never valid and thus can never result in a valid page mapping. They

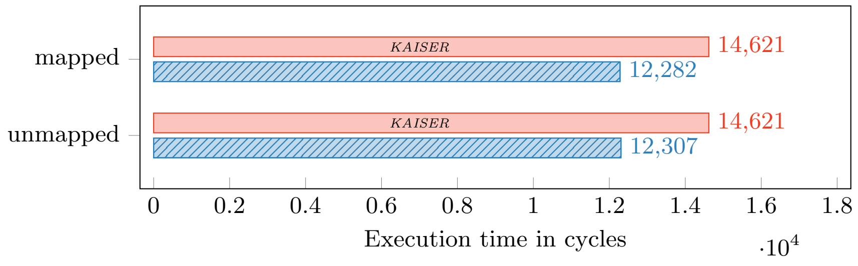


Fig. 4: Double page fault attack with and without *KAISER*: mapped and unmapped pages cannot be distinguished if *KAISER* is in place.

are commonly referred to as segmentation faults, typically terminating the user program.

Intel TSX-based Attack. The Intel TSX-based attack presented by Jang et al. [10] (cf. Section 2) exploits the same timing difference as the double page fault attack. However, with Intel TSX the page fault handler is not invoked, resulting in a significantly faster and more stable attack. As the basic underlying principle is equivalent to the double page fault attack, *KAISER* successfully prevents this attack as well. Figure 5 shows the execution time of a TSX transaction for unmapped pages, non-executable mapped pages, and executable mapped pages. With the default kernel, the transaction execution time is 299 cycles for unmapped pages, 270 cycles for non-executable mapped pages, and 226 cycles for executable mapped pages. With *KAISER*, we measure a constant timing of 300 cycles. As in the double page fault attack, *KAISER* successfully eliminates the timing side channel.

We also verified this result by running the attack demo by Jang et al. [9]. On the default kernel, the attack recovers page mappings with a 100 % accuracy. With *KAISER*, the attack does not even detect a single mapped page and consequently no modules.

Prefetch Side-Channel Attack. As described in Section 2, prefetch side-channel attacks exploit timing differences in software prefetch instructions to obtain address information. We evaluate the efficacy of *KAISER* against the two prefetch side-channel attacks presented by Gruss et al. [6].

Figure 6 shows the median execution time of the `prefetch` instruction in cycles compared to the actual address translation level. We observed an execution time of 241 cycles on our test system for page translations terminating at PDPT level and PD level respectively. We observed an execution time of 237 cycles when the page translation terminates at the PT level. Finally, we observed a distinct execution times of 212 when the page is present and cached, and 515 when the page is present but not cached. As in the previous attack, *KAISER* successfully eliminates any timing differences. The measured execution time is 241 cycles in all cases.

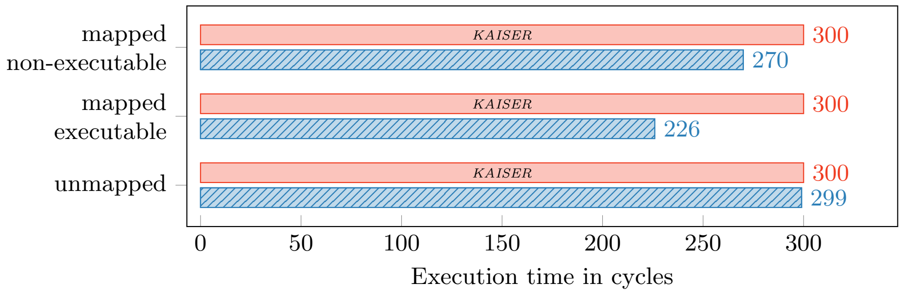


Fig. 5: Intel TSX-based attack: On the default kernel, the status of a page can be determined using the TSX-based timing side channel. *KAISER* completely eliminates the timing side channel, resulting in an identical execution time independent of the status.

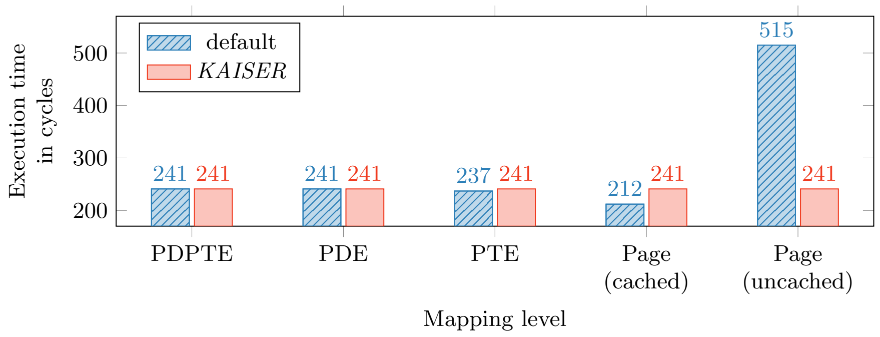


Fig. 6: Median prefetch execution time in cycles depending on the level where the address translation terminates. With the default kernel, the execution time leaks information on the translation level. With *KAISER*, the execution time is identical and thus does not leak any information.

Figure 7 shows the address-translation attack. While the correct guess can clearly be detected without the countermeasure (dotted line), *KAISER* eliminates the timing difference. Thus, the attacker is not able to determine the correct virtual-to-physical translation anymore.

4.2 Performance Evaluation

As described in Section 3.2, *KAISER* has a low memory overhead of 8 kB per user thread, 12 kB per user process, and a system-wide total overhead of 1 MB. A full-blown Ubuntu Linux already consumes several hundred megabytes of memory. Hence, in our evaluation the memory overhead introduced by *KAISER* was hardly observable.

In order to evaluate the runtime performance impact of *KAISER*, we execute different benchmarks with and without the countermeasure. We use the PARSEC

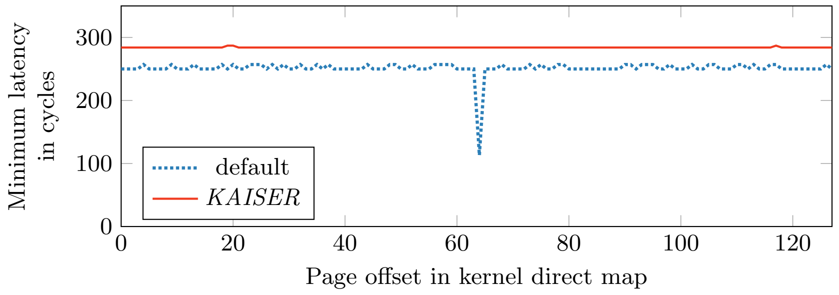


Fig. 7: Minimum access time after prefetching physical direct-map addresses. The low peak in the dotted line reveals to which physical address a virtual address maps (running the default kernel). The solid line shows the same attack on a kernel with *KAISER* active. *KAISER* successfully eliminates the leakage.

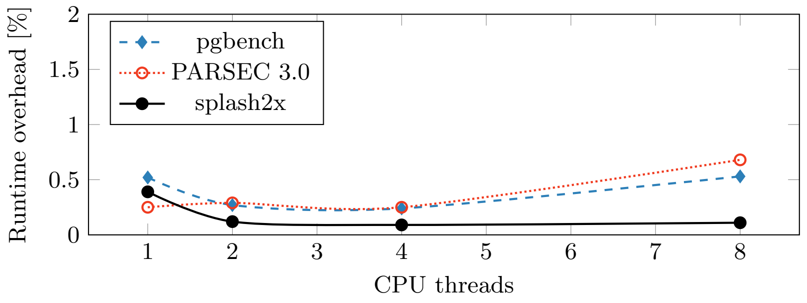


Fig. 8: Comparison of the runtime of different benchmarks when running on the *KAISER*-protected kernel. The default kernel serves as baseline (=100%). We see that the average overhead is 0.28% and the maximum overhead is 0.68%.

3.0 [1] (input set "native"), the pgbench [24] and the SPLASH-2x [16] (input set "native") benchmark suites to exhaustively measure the performance overhead of *KAISER* in various different scenarios.

The results of the different benchmarks are summarized in Figure 8 and Table 1. We observed a very small average overhead of 0.28% for all benchmark suites and a maximum overhead of 0.68% for single tests. This surprisingly low performance overhead underlines that *KAISER* should be deployed in practice.

4.3 Reproducibility of Results

In order to make our evaluation of efficacy and performance of *KAISER* easily reproducible, we provide the source code and precompiled Debian packages compatible with Ubuntu 16.10 on GitHub. The repository can be found at `https://github.com/IAIK/KAISER`. We fully document how to build the Ubuntu Linux kernel with *KAISER* protections from the source code and how to obtain the benchmark suites we used in this evaluation.

Table 1: Average performance overhead of *KAISER*.

Benchmark	Kernel	Runtime				Average Overhead
		1 core	*2 cores*	*4 cores*	*8 cores*	
PARSEC 3.0	default	27:56,0 s	14:56,3 s	8:35,6 s	7:05,1 s	0.37 %
	KAISER	28:00,2 s	14:58,9 s	8:36,9 s	7:08,0 s	
pgbench	default	3:22,3 s	3:21,9 s	3:21,7 s	3:53,5 s	0.39 %
	KAISER	3:23,4 s	3:22,5 s	3:22,3 s	3:54,7 s	
SPLASH-2X	default	17:38,4 s	10:47,7 s	7:10,4 s	6:05,3 s	0.09 %
	KAISER	17:42,6 s	10:48,5 s	7:10,8 s	6:05,7 s	

5 Future Work

KAISER does not consider BTB attacks, as they require knowledge of the BTB behavior. The BTB behavior has not yet been reverse-engineered for recent Intel processors, such as the Skylake microarchitecture (cf. Section 2.3). However, if the BTB is reverse-engineered in future work, attacks on systems protected by *KAISER* would be possible. Evtyushkin et al. [3] proposed to use virtual address bits > 30 to randomize memory locations for KASLR as a zero-overhead countermeasure against BTB attacks. *KAISER* could incorporate this adaption to effectively mitigate BTB attacks as well.

Intel x86-64 processors implement multiple features to improve the performance of address space switches. Linux currently does not make use of all features, e.g., Linux could use process-context identifiers to avoid some TLB flushes. The performance of *KAISER* would also benefit from these features, as *KAISER* increases the number of address space switches. Consequently, utilizing these optimization features could lower the runtime overhead below 0.28%.

KAISER exploits very recent processor features which are not present on older machines. Hence, we expect higher overheads on older machines if *KAISER* is employed for security reasons. The current proof-of-concept implementation of *KAISER* shows that defending against the attack is possible. However, it does not eliminate all KASLR information leaks, especially information leaks that are not caused by the same hardware effects. A full implementation of *KAISER* must map any randomized memory locations that are used during the context switch at fixed offsets. This is straightforward, as we have already introduced new mappings which can easily be extended. During the context switch, kernel memory locations are only accessed through these fixed mappings. Hence, the offsets of the randomized parts of the kernel can not be leaked in this case.

6 Conclusion

In this paper we discussed limitations of x86 impeding practical kernel address isolation. We show that our countermeasure (*KAISER*) overcomes these limitations and eliminates all microarchitectural side-channel attacks on kernel address information on recent Intel Skylake systems. More specifically, we show that *KAISER* protects the kernel against double page fault attacks, prefetch side-channel attacks, and TSX-based side-channel attacks. *KAISER* enforces a strict kernel and user space isolation such that the hardware does not hold any information about kernel addresses while running user processes. Our proof-of-concept is implemented on top of a full-fledged Ubuntu Linux kernel. *KAISER* has a low memory overhead of approximately 8 kB per user thread and a low runtime overhead of only 0.28%.

Acknowledgments

We would like to thank our anonymous reviewers, Anders Fogh, Rodrigo Branco, Richard Weinbeger, Thomas Garnier, David Gens and Mark Rutland for their valuable feedback. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). This work was partially supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments".

References

1. Bienia, C.: Benchmarking Modern Multiprocessors. Ph.D. thesis, Princeton University (Jan 2011)
2. Branco, R., Gueron, S.: Blinded random corruption attacks. In: IEEE International Symposium on Hardware Oriented Security and Trust (HOST'16) (2016)
3. Evtyushkin, D., Ponomarev, D., Abu-Ghazaleh, N.: Jump over aslr: Attacking branch predictors to bypass aslr. In: International Symposium on Microarchitecture (MICRO'16) (2016)
4. Gras, B., Razavi, K., Bosman, E., Bos, H., Giuffrida, C.: ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS'17 (2017)
5. Gruss, D., Maurice, C., Mangard, S.: Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA'16 (2016)
6. Gruss, D., Maurice, C., Fogh, A., Lipp, M., Mangard, S.: Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS'16 (2016)
7. Hund, R., Willems, C., Holz, T.: Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P'13 (2013)
8. Intel: Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide 253665 (2014)
9. Jang, Y.: The DrK Attack - Proof of concept. `https://github.com/sslab-gatech/DrK` (2016), retrieved on February 24, 2017

10. Jang, Y., Lee, S., Kim, T.: Breaking Kernel Address Space Layout Randomization with Intel TSX. In: CCS'16 (2016)
11. Kemerlis, V.P., Polychronakis, M., Keromytis, A.D.: ret2dir: Rethinking kernel isolation. In: USENIX Security Symposium. pp. 957–972 (2014)
12. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In: ISCA'14 (2014)
13. Kirill A. Shutemov: Pagemap: Do Not Leak Physical Addresses to Non-Privileged Userspace. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ab676b7d6fbf4b294bf198fb27ade5b0e865c7ce` (Mar 2015), retrieved on November 10, 2015
14. Levin, J.: Mac OS X and IOS Internals: To the Apple's Core. John Wiley & Sons (2012)
15. Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Boano, C.A., Mangard, S., Römer, K.: Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS'17 (2017), to appear
16. PARSEC Group: A Memo on Exploration of SPLASH-2 Input Sets. `http://parsec.cs.princeton.edu` (2011)
17. PaX Team: Address space layout randomization (aslr). `http://pax.grsecurity.net/docs/aslr.txt` (2003)
18. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., Mangard, S.: DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security Symposium (2016)
19. Russinovich, M.E., Solomon, D.A., Ionescu, A.: Windows internals. Pearson Education (2012)
20. Seaborn, M., Dullien, T.: Exploiting the DRAM rowhammer bug to gain kernel privileges. In: Black Hat 2015 Briefings (2015)
21. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: 14th ACM CCS (2007)
22. Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: CCS'04 (2004)
23. Solar Designer: Getting around non-executable stack (and fix). `http://seclists.org/bugtraq/1997/Aug/63` (Aug 1997)
24. The PostgreSQL Global Development Group: pgbench. `https://www.postgresql.org/docs/9.6/static/pgbench.html` (2016)
25. Venkatasubramanian, G., Figueiredo, R.J., Illikkal, R., Newell, D.: TMT: A TLB tag management framework for virtualized platforms. International Journal of Parallel Programming 40(3) (2012)