SECURITY

Kernel Isolation
From an Academic Idea to an Efficient Patch for Every Computer

DANIEL GRUSS, DAVE HANSEN, AND BRENDAN GREGG

Daniel Gruss (@lavados) is a postdoc fellow at Graz University of Technology. He has been involved with teaching at the university since 2010. In 2015, he demonstrated Rowhammer.js, the first remote fault attack running in a website. He was part of the research team that found the Meltdown and Spectre bugs published in early 2018. daniel.gruss@iaik.tugraz.at

Dave Hansen works in Intel's Open Source Technology Center in Hillsboro, Oregon. He has been involved in Linux for over 15 years and has worked on side-channel hardening, scalability, NUMA, memory management, and many other areas. dave.hansen@intel.com

Brendan Gregg is an industry expert in computing performance and cloud computing. He is a Senior Performance Architect at Netflix, where he does performance design, evaluation, analysis, and tuning. He is the author of Systems Performance published by Prentice Hall, and he received the USENIX LISA Award for Outstanding Achievement in System Administration. Brendan has created performance analysis tools included in multiple operating systems, and visualizations and methodologies for performance analysis, including flame graphs. bgregg@netflix.com

The disclosure of the Meltdown vulnerability [9] in early 2018 was an earthquake for the security community. Meltdown allows temporarily bypassing the most fundamental access permissions before a deferred permission check is finished: that is, the userspace-accessible bit is not reliable, allowing unrestricted access to kernel pages. More specifically, during out-of-order execution, the processor fetches or stores memory locations that are protected via access permissions and continues the out-of-order execution of subsequent instructions with the retrieved or modified data, *even if the access permission check failed*. Most Intel, IBM, and Apple processors from recent years are affected as are several other processors. While AMD also defers the permission check, it does not continue the out-of-order execution of subsequent instructions with data that is supposed to be inaccessible.

KAISER [4, 5] was designed as a software-workaround to the userspace-accessible bit. Hence, KAISER eliminates any side-channel timing differences for inaccessible pages, making the hardware bit mostly superfluous. In this article, we discuss the basic design and the different patches for Linux, Windows, and XNU (the kernel in modern Apple operating systems).

Basic Design

Historically, the kernel was mapped into the address space of every user program, but kernel addresses were not accessible in userspace because of the userspace-accessible bit. Conceptually, this is a very compact way to define two address spaces, one for user mode and one for kernel mode. The basic design of the KAISER mechanism and its derivates is based on the idea that the userspace-accessible bit is not reliable during transient out-of-order execution. Consequently, it becomes necessary to work around this permission bit and not rely on it.

As shown in Figure 1, we try to emulate what the userspace-accessible bit was supposed to provide, namely two address spaces for the user program: a kernel address space with all addresses mapped, protected with proper use of SMAP, SMEP, and NX; and a user address space that only includes a very small fraction of the kernel. This small fraction is required due to the way context switches are defined on the x86 architecture. However, immediately after switching into kernel mode, we switch from the user address space to the kernel address space. Thus, we only have to make sure that read-only access to the small fraction of the kernel does not pose a security problem.

As we discuss in more detail in the performance section, emulating the userspace-accessible bit through this hard split of the address spaces comes with a performance cost.

The global bit. As page table lookups can take much time, a multi-level cache hierarchy (the translation lookaside buffer, TLB) is used to improve the performance. When switching between processes, the TLB has to be cleared at least partially. Most operating systems optimize the performance of context switches by using the global bit for TLB entries that are also valid in the next address space. Consequently, we have to use it with care when implementing

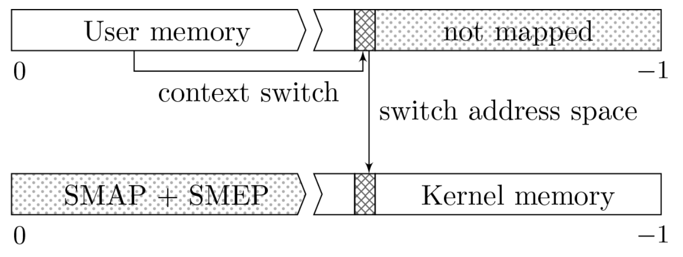


Figure 1: The basic KAISER mechanism

the design outlined above. In particular, marking kernel pages as global (as operating systems previously did) completely undermines the security provided by the KAISER mechanism. Setting the bit to 0 eliminates this problem but leads to another performance reduction.

Naming the Patches. The name KAISER is supposed to be an acronym for *Kernel Address Isolation to have Side channels Efficiently Removed*. It is also a reference to the emperor penguin (German: "Kaiserpinguin"), the largest penguin on earth, with the penguin being the Linux mascot and KAISER being a patch to make Linux stronger. Still under the name KAISER, a significant amount of work was put into the patches that we outline later in this article. Both the authors of the KAISER patch and the Linux kernel maintainers also discussed other names that were deemed less appropriate. Shortly before merging KAISER into the mainline kernel, it was renamed to KPTI, which fits in the typical Linux naming scheme.

Naturally, Microsoft and Apple could not just copy either of the names of the Linux patch. Consequently, they came up with their own names (i.e., KVA Shadow and Double Map) for their own variants of the same idea.

Actual Implementations

The KAISER implementation, developed mainly on virtual machines and a specific off-the-shelf Skylake system, focused on proving that the basic approach was sound. Consequently, reliability and stability that would allow deployment in a real-world environment were out of scope for KAISER. Bringing KAISER up to industry and community standards required ensuring support for all existing hardware and software features and improving its performance and security properties. Furthermore, for Windows and XNU, the patches had to be redeveloped from scratch since their design and implementation is substantially different from Linux.

While the focus on specific machine environments limited the scope of the effort and enabled the implementation of a rapid proof of concept, the environment did not have to cope with certain hardware features like non-maskable interrupts (NMIs), or corner cases when entering or exiting the kernel. These corner cases are rarely encountered in the real world but must still be handled because they might be exploited to cause crashes or escalate privileges (e.g., CVE-2014-4699). NMIs are a particular challenge because they can occur in almost any context, including while the kernel is attempting to transition to or from userspace. For example, before the kernel attempts to return from an interrupt to userspace, it first switches to the user address space. At least one instruction later, it actually transitions to userspace. This means there is always a window where the kernel appears to be running with the "wrong" address space. This can confuse the address-space-switching code, which must use a different method to determine which address space to restore when returning from the NMI.

Linux's KPTI

Much of the process of building on the KAISER proof of concept (PoC) was iterative: find a test that fails or crashes the kernel, debug, fix, check for regressions, then move to the next test. Fortunately, the "x86 selftests" test many infrequently used features, such as the `modify ldt` system call, which is rarely used outside of DOS emulators. Virtually all of these tests existed before KAISER. The key part of the development was finding the tests that exercised the KAISER-impacted code paths and ensuring the tests got executed in a wide variety of environments.

KAISER focused on identifying all of the memory areas that needed to be shared by the kernel and user address spaces and mapping those areas into both. Once it neared being feature-complete and fully functional, the focus shifted to code simplification and improving security.

The shared memory areas were scattered in the kernel portion of the address space. This led to a complicated kernel memory map that made it challenging to determine whether a given mapping was correct, or might have exposed valuable secrets to an application. The solution to this complexity is a data structure called `cpu_entry_area`. This structure maps all of the data and code needed for a given CPU to enter or exit the kernel. It is located at a consistent virtual address, making it simple to use in the restricted environment near kernel entry and exit points. The `cpu_entry_area` is strictly an alias for memory mapped elsewhere by the kernel. This allows it to have hardened permissions for structures such as the "task state segment," mapping them read-only into the `cpu_entry_area` while still permitting the other alias to be used for modifications.

While the kernel does have special "interrupt stacks," interrupts and **system call** instructions still use a process's kernel stack for a short time after entering the kernel. For this reason, KAISER mapped all process kernel stacks into the user address space. This potentially exposes the stack contents to Meltdown, and it also creates performance overhead in the `fork()` and `exit()` paths. To mitigate both the performance and attack exposure, KPTI added special "entry stacks" to the `cpu_entry_area`. These stacks are only used for a short time during kernel entry/exit and contain much more limited data than the full process stack, limiting the likelihood that they might contain secrets.

Historically, any write to the `CR3` register invalidates the contents of the TLB, which has hundreds of entries on modern processors. It takes a significant amount of processor resources to replace these contents when frequent kernel entry/exits necessitate frequent `CR3` writes. However, a feature on some x86 processors, Process Context Identifiers (PCIDs), provides a mechanism to allow TLB entries to persist over `CR3` updates. This allows TLB contents to be preserved over system calls and interrupts, greatly reducing the TLB impact from `CR3` updates [6]. However, allowing multiple address spaces to live within the TLB simultaneously requires additional work to track and invalidate these entries. But the advantages of PCIDs outweigh the disadvantages, and it continues to be used in Linux both to accelerate KPTI and to preserve TLB contents across normal process context-switching.

Microsoft Windows' KVA Shadow

Windows introduced Kernel Virtual Address (KVA) Shadow mapping [7], which follows the same basic idea as KAISER, with necessary adaptations to the Windows operating system. However, KVA Shadow does not have the goal of ensuring the robustness of KASLR in general, but only seeks to mitigate Meltdown-style attacks. This is a deliberate design choice made to avoid unnecessary design complexity of KVA Shadow.

Similar to Linux, KVA Shadow tries to minimize the number of kernel pages that remain mapped in the user address space. This includes hardware-required per-processor data and special per-processor transition stacks. To not leak any kernel information through these transition stacks, the context switching code keeps interrupts disabled and makes sure not to trigger any kernel traps.

The significant deviations from the basic KAISER approach are in the performance optimizations implemented to make KVA Shadow practical for the huge Windows user base. Similar to Linux, this included the use of PCIDs to minimize the number of implicit TLB flushes. Another interesting optimization is "user/global acceleration" [7]. As stated in the Basic Design section, above, the global bit tells the hardware whether or not to keep TLB entries across the next context switch. While the global bit can no longer be used for kernel pages, Windows now uses it for user pages. Consequently, switching from user to kernel mode does not flush the user TLB entries, although the `CR3` register is switched. This yields a measurable performance advantage. The user pages are not marked global in the kernel address space, and, hence, the corresponding TLB entries are correctly invalidated during the context switch to the next process.

Windows further optimizes the execution of highly privileged tasks by letting them run with a conventional shared address space, which is identical to what the "kernel" address space is now.

With a large number of third-party drivers and software deeply rooted in the system (e.g., anti-viruses), it is not unexpected that some contained code assumes a shared address space. While this first caused compatibility problems, subsequent updates resolved these issues.

Apple XNU's Double Map

Apple introduced the Double Map feature in macOS 10.13.2 (i.e., XNU kernel 4570.31.3, Darwin 17.3.0). Apple used PCIDs on x86 already in earlier macOS versions. However, because mobile Apple devices are also affected by Meltdown, mitigations in the ARMv8-64 XNU kernel were required. Here Apple introduced an interesting technique to leverage the two Translation Table Base Registers (TTBRs) present on ARMv8-64 cores and the Translation Control Register (TCR), which controls how the TTBRs are used in the address translation.

The virtual memory is split into two halves, a userspace half mapped via TTBR0 and a kernel space half mapped via TTBR1. The TCR allows splitting the address space and assigning different TTBRs to disjoint address space ranges. Apple's XNU kernel uses the TCR to unmap the protected part of the kernel in user mode. That is, the kernel space generally remains mapped in every user process, but it's unmapped via the TCRs when leaving the kernel. Kernel parts required for the context switch, interrupt entry code, and data structures are below a certain virtual address and remain mapped. When entering the kernel again, the kernel reconfigures the address space range of TTBR1 via the TCR and, by that, remaps the protected part of the kernel.

The most important advantage of this approach is that the translation tables are not duplicated or modified while running in user mode. Hence, any integrity mechanisms checking the translation tables continue to work.

Performance

When publishing the first unstable PoC of KAISER, the question of performance impact was raised. While the performance impact was initially estimated to be below 5% [5], KAISER showed once more how difficult it is to measure performance in a way that allows comparison of performance numbers. With PCIDs or ASIDs, as now used by all major operating systems, the performance overheads of the different real-world KAISER implementations were reduced, but there are still overheads that may be significant, depending on the workload and the specific hardware. Still, the performance loss for different use cases,

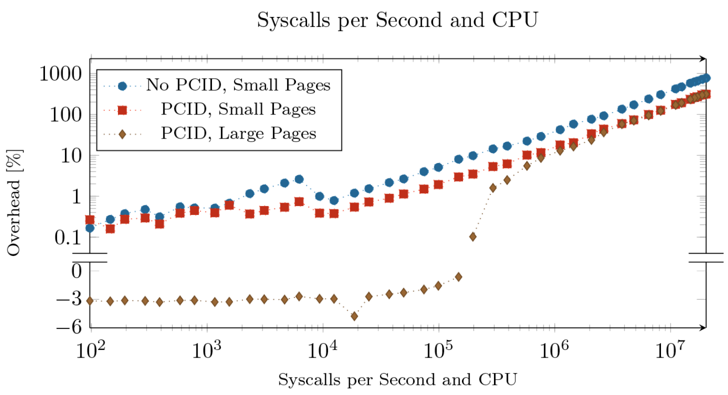


Figure 2: The runtime overhead for different workloads with different KPTI configurations [2]. The overhead increases with the system call rate due to the additional TLB flushes and CR3 manipulations during context switches.

macrobenchmarks, and microbenchmarks varies between −5% and 800%. One reason is the increase in TLB flushes, especially on systems without PCID support, as well as extra cycles for CR3 manipulation. More indirect is the increase in TLB pressure, caused by the additional TLB entries due to the large number of duplicated page table entries. CPU- or GPU-intense workloads that trigger a negligible number of context switches, and thus a negligible number of TLB flushes and CR3 manipulations, are mostly unaffected.

The different implementations of KAISER have different optimizations. In this performance analysis, we focus on Linux (i.e., KPTI). However, the reported numbers are well aligned with reports of performance overheads on other operating systems [1, 7].

We explore the overheads for different system call rates [2] by timing a simultaneous working-set walk, as shown in Figure 2.

Without PCID, at low system call rates, the overheads were negligible, as expected: near 0%. At the other end of the spectrum, at over 10 million system calls per second per CPU, the overhead was extreme: the benchmark ran over 800% slower. While it is unlikely that a real-world application will come anywhere close tonthis, it still points out a relevant bottleneck that has not existed without the KAISER patches. For perspective, the system call rates for different cloud services at Netflix were studied, and it was found that database services were the highest, with around 50,000 system calls per second per CPU. The overhead at this rate was about 2.6% slower.

While PCID support greatly reduced the overhead, from 2.6% to 1.1%, there is another technique to reduce TLB pressure: large pages. Using large pages reduces the overhead for our specific benchmark so much that for any real-world system call rate there is a performance gain.

Another interesting observation while running the microbenchmarks was an abrupt drop in performance overhead, depending on the hardware and benchmark, at a syscall rate of 5000. While this was correlated with the last-level cache hit ratio, it is unclear what the exact reason is. One suspected cause is a sweet spot in either the amount of memory touched or the access pattern between two system calls, where, for example, the processor switches the cache eviction policy [3].

With PCID support and using large pages when possible, one can conclude that the overheads of Linux's KPTI and other KAISER implementations are acceptable. Furthermore, rudimentary performance tuning (i.e., analyzing and reducing system call and context switch rates) may yield additional performance gains.

Outlook and Conclusion

With KAISER and related real-world patches, we accepted a performance overhead to cope with the insufficient hardware-based isolation. While more strict isolation can be a more resilient design in general, it currently functions as a workaround for a specific hardware bug. However, there are more Meltdown-type hardware bugs [8, 10], causing unreliable permission checks during transient out-of-order execution, for other page table bits. Mitigating them requires additional countermeasures beyond KAISER. For now, KAISER will still be necessary for commodity processors.

Acknowledgments

We would like to thank Matt Miller, Jon Masters, and Jacques Fortier for helpful comments on early drafts of this article.

References

[1] fG!, "Measuring OS X Meltdown Patches Performance," January 2018: https://reverse.put.as/2018/01/07/measuring-osx-meltdown-patches-performance/.

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

[3] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '16)*, pp. 300–321: https://gruss.cc/files/rowhammerjs.pdf.

[4] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR," in 23rd ACM Conference on Computer and Communications Security (CCS, 2016): https://gruss.cc/files/prefetch.pdf.

[5] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR Is Dead: Long Live KASLR," in *Proceedings of the 9th International Symposium on Engineering Secure Software and Systems (ESSoS '17)*, pp.161–176: https://gruss.cc/files/kaiser.pdf.

[6] D. Hansen, "KAISER: Unmap Most of the Kernel from User-space Page Table," Linux Kernel Mailing List, October 2017: https://lkml.org/lkml/2017/10/31/884.

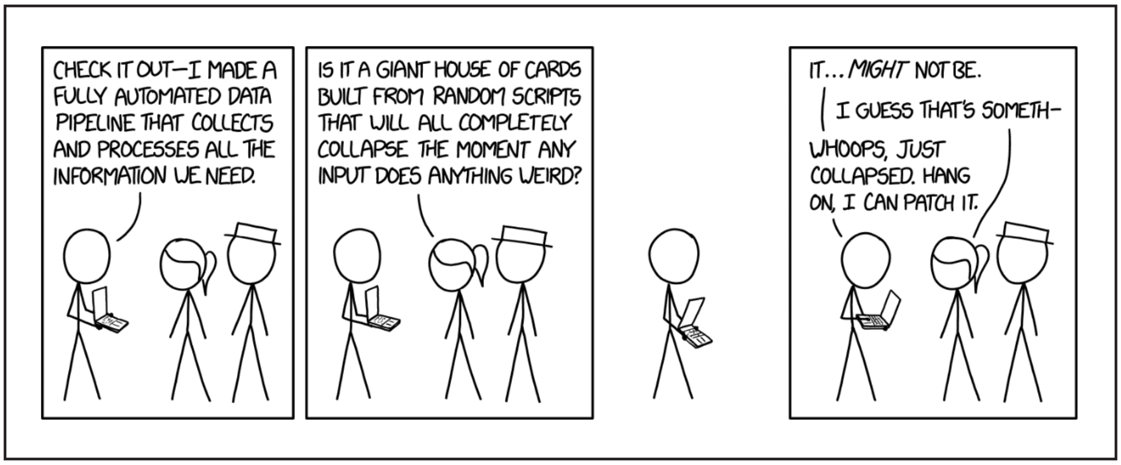
[7] K. Johnson, "KVA Shadow: Mitigating Meltdown on Windows," March 2018: https://blogs.technet.microsoft.com/srd/2018/03/23/kva-shadow-mitigating-meltdown-on-windows/.

[8] V. Kiriansky and C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses," *arXiv:1807.03757*, 2018.

[9] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security '18)*, pp. 973–990: https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-lipp.pdf.

[10] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security '18)*, pp. 991–1008: https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-van_bulck.pdf.

XKCD xkcd.com


CHECK IT OUT—I MADE A FULLY AUTOMATED DATA PIPELINE THAT COLLECTS AND PROCESSES ALL THE INFORMATION WE NEED.
IS IT A GIANT HOUSE OF CARDS BUILT FROM RANDOM SCRIPTS THAT WILL ALL COMPLETELY COLLAPSE THE MOMENT ANY INPUT DOES ANYTHING WEIRD?
IT... *MIGHT* NOT BE.
I GUESS THAT'S SOMETH–
WHOOPS, JUST COLLAPSED. HANG ON, I CAN PATCH IT.