Dealing with (Some of) the Fallout from Meltdown

Nadav Amit
VMware Research
namit@vmware.com

Michael Wei
VMware Research
mwei@vmware.com

Dan Tsafrir
VMware Research & Technion
dtsafrir@vmware.com

ABSTRACT

The meltdown vulnerability allows users to read kernel memory by exploiting a hardware flaw in speculative execution. Processor vendors recommend "page table isolation" (PTI) as a software fix, but PTI can significantly degrade the performance of system-call-heavy programs. Leveraging the fact that 32-bit pointers cannot access 64-bit kernel memory, we propose "Shrink", a safe alternative to PTI, which is applicable to programs capable of running in 32-bit address spaces. We show that Shrink can restore the performance of some workloads, suggest additional potential alternatives, and argue that vendors must be more open about hardware flaws to allow developers to design protection schemes that are safe *and* performant.

CCS CONCEPTS

• **Security and privacy** → **Operating systems security**.

ACM Reference Format:
Nadav Amit, Michael Wei, and Dan Tsafrir. 2021. Dealing with (Some of) the Fallout from Meltdown. In *The 14th ACM International Systems and Storage Conference (SYSTOR '21), June 14–16, 2021, Haifa, Israel.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3456727.3463776

1 INTRODUCTION

The "meltdown" hardware security vulnerability enables unprivileged processes to read inaccessible kernel memory by exploiting speculative execution. Roughly speaking, a malicious user can trick the CPU into speculatively accessing u_j, such that u is a user array and j is some private data of the kernel that is unknown to the user. Because u_j is now cached in user-space, the user can deduce the value of j by timing access to u's elements.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SYSTOR '21, June 14–16, 2021, Haifa, Israel
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8398-1/21/06...$15.00
https://doi.org/10.1145/3456727.3463776

Meltdown was disclosed to some researchers and vendors as early as July 2017, and later, in January 2018, it was disclosed to the general public [27].The vulnerability affects Intel, IBM, and ARM processors released over the last two decades. Fixing this vulnerability will have a cost in terms of real dollars that may eclipse the Y2K bug, estimated as over $100 billion in the US alone [28]. Unlike Y2K, fixing meltdown will have a lasting performance impact, as patching it requires establishing barriers to speculation and isolating the kernel.

Meltdown was made possible because operating systems (OSes) traditionally map the kernel's address space in the page table of *every* process for efficiency. System designers rely on hardware protection to prevent unauthorized user access by marking the kernel memory pages as privileged. Unfortunately, on meltdown-vulnerable CPUs, a user process can speculatively access these privileged kernel pages, leaking kernel data indirectly.

The canonical defense against meltdown recommended by CPU vendors is to separate the kernel and user into two different address spaces. This technique, known as "page table isolation" (PTI) [11], is employed in BSD [31], Linux [21], OS X [6], and Windows [25].Unfortunately, PTI has been shown to reduce the performance of some workloads by 30% or more [22]. Especially affected are workloads that frequently make system calls into the kernel and must therefore suffer PTI overheads associated with context switching. Intel has mitigated Meltdown in hardware in newer silicon [9], which no longer requires PTI. However, there are billions of processors still in service, and Intel is still producing silicon with the vulnerabilities, which can be purchased in new products even today, three years after Meltdown's public release [2].

Meltdown mitigations are especially dire for embedded, real-time applications which use meltdown-vulnerable processors, such as avionics [7], railway controls [17], medical [29] and industrial controls [30]. These safety-critical systems are deployed with the expectation that the processor would operate in a fixed performance envelope, which might be lost when PTI is enabled [8]. In § 2, we discuss meltdown and PTI in detail.

Motivated by PTI overheads, we propose an alternative [3], called *Shrink*, which is only applicable to workloads fitting into 32-bit address spaces. Shrink relies on 32-bit compatibility hardware feature to provide protection: in a 64-bit system, the kernel space resides outside of any 32-bit space, so Shrink

is able to prevent the CPU from speculatively reading kernel pages (§ 3). Resorting to 32-bit compatibility mode may, in general, hamper performance. Nevertheless, we show that system-call-heavy workloads that do little computation benefit from Shrink and perform nearly as good as a system without PTI (§ 4). A user can easily check if her application benfits by simply running the corresponding 32-bit version.

In addition to Shrink, we enumerate several other mitigations, which we discuss but not evaluate (§ 5). These defenses include leveraging memory segmentation, binary translation, compiler checks, and dedicating cores to the kernel. While we submitted patches implement Shrink to the Linux kernel, these patches were ultimately rejected, and we discuss the reasons for rejection as well as how Shrink and its ideas may still benefit the community (§ 6).

2 MELTDOWN AND DEFENDING AGAINST IT

Meltdown exploits the speculative execution mechanism of the processor, tricking it to access privileged data [23]. While speculative execution correctly prevents unprivileged processes from directly reading the speculative results, speculation has side-effects that can be observed and exploited by unprivileged processes. The meltdown proof-of-concept is able to deduce privileged data using timing attacks against data left in the cache. Future potential exploits could perhaps use other side-channels (such as counters or power). Thus, a robust defense against meltdown should eliminate speculative execution on privileged data completely.

Meltdown affects several processor architectures (ARM, Power, and x86). This paper focuses on x86.

Let P be a kernel memory page that an unprivileged process U is forbidden to access. U may speculatively read P only if U has P mapped in its page table, which means there exists a virtual memory address V that U can use to refer to P. Before meltdown (Figure 1a), V existed, because OSes typically mapped P (along with the entire system memory) in *each* page table of *each* process. For protection, OSes relied on hardware, marking P as "supervisor" in its PTE (page table entry), thereby instructing the CPU to allow only the kernel to access P. OSes additionally marked P as "global", which means that the $P \Rightarrow V$ mapping remains valid across context switches in the TLB (translation lookaside buffer), a vital CPU cache that accelerates virtual-to-physical translations.

Meltdown-vulnerable CPUs defer checking the supervisor bit, allowing cores to speculatively execute using privileged mappings and thereby opening a side-channel that U might exploit as outlined in § 1. Kernel page table isolation (PTI) [11] addresses the problem by removing most kernel mappings from user space and maintaining a second, fuller address space to be used by the kernel when servicing U (Figure 1b).

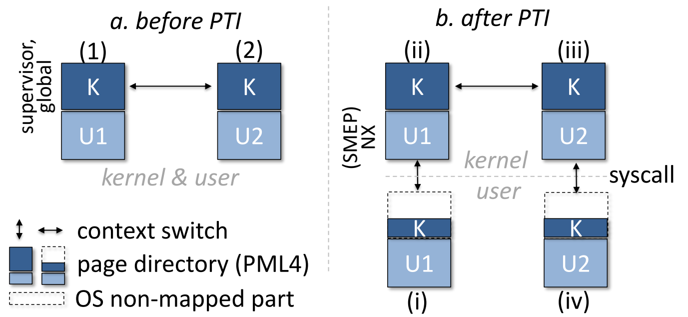

Figure 1: *Traditionally, the OS handled system calls while running in the address space of the calling process (left). With PTI, every system call requires a context switch (right).*

PTI attempts to reduce its overheads, which can nevertheless be prohibitive [12, 20]. The exact performance penalty of PTI partially depends on features supported by the CPU. Next, we describe PTI in more detail and highlight some of its overhead components and the CPU features that affect them.

In x86, an address space corresponds to one PML4, a per-process 4KB-page that serves as the root of a radix tree page-table hierarchy that translates virtual to physical addresses. The per-core CR3 register holds the physical address of the PML4 of the currently running process. A context switch occurs when CR3 is assigned a new address, causing non-global TLB entries to be flushed. Before PTI, the OS kept CR3 as is when servicing system calls (or interrupts); CR3 changed only when switching from one process to another, and even then global kernel mappings remained valid in the TLB (Figure 1a). In contrast, with PTI, the OS updates CR3 upon each kernel entry and exit, for every system call (Figure 1b), and no global mappings are used to defend against meltdown. The user's PML4 (R_u) only allows access to the minimal kernel pages needed, notably to enter the kernel ("trampoline") to perform interrupt handling and system call entry. The kernel's per-process PML4 (R_k) encompasses both user and kernel mappings.

PTI kernel-user PML4-s are physically contiguous, so switching between them can be done by flipping only one CR3 bit, without having to map potentially sensitive OS data. The two PML4-s point to the same user page tables, so that updates need only be applied once, albeit TLB invalidations must be applied twice. Updating mappings is expensive, not only in terms of the direct cost of flushing and updating the TLB, but also due to TLB misses that occur as a result from having different kernel mappings in each address space. For security, user pages mapped in R_k are non-executable using SMEP (supervisor mode execution protection). In CPUs without SMEP support, PTI may emulate SMEP functionality by setting the NX (no execute) bits in the corresponding PTEs.

Newer x86 processors (on Intel, since Westmere, 2010), include the "process context identifier" (PCID) feature, which

enables OSes to tag mappings with an identifier. Only mappings with a PCID that matches that of the current process are used, eliminating the need to flush the entire TLB upon context switching. The initial implementation of PCID, however, only allows flushing TLB entries of the current address space, whereas PTI must flush for both R_u and R_k, so it resorts to a full TLB flush. A mechanism to remove mappings with a specific PCID (the INVPCID instruction) was not added until Haswell (2013), whose server versions (Haswell-EP / Haswell-EX) were not introduced until September 2014. Regardless, with PTI, when a *kernel* mapping is changed, all PCIDs must be flushed, so a full TLB flush is required [15].

In § 4 we evaluate the cost of PTI across various microarchitectures. We find that PTI increases the cost of a system call by 4× on a Skylake and Haswell with INVPCID support, which increases to nearly 8× for a Nehalem (2008), which has no PCID support.

Virtual machines (VMs) update CR3 as if running natively. Most recent Intel microarchitectures (since Nehalem) allow that with the "extended page table" (EPT) feature, which provides secondary page table management for guests without host involvement. Running VMs without EPT triggers exits to the host whenever CR3 is updated, which greatly increases the penalty of PTI, forcing a VM-exit on every user-kernel transition. The result is that system calls are ~15× more expensive under PTI. Intel still ships processors, such as the Pentium A1020 and Atom E680T, a popular embedded processor) with virtualization support (VT-x) but no EPT support.

3 SHRINK WITH 32-BIT COMPATIBILITY

Shrink uses the 32-bit compatibility mode feature of x86-64 architecture to contain the accesses of a userspace process, even speculatively, into a 32-bit space. The kernel pages are mapped outside of this 32-bit space so that the 32-bit process has no mechanism to address kernel mappings since it is confined to 32-bit registers. Shrink avoids PTI penalties by protecting against meltdown speculative accesses through shrinking the user addressable process space rather than mapping/unmapping the kernel.

Running in 32-bit compatibility mode has significant drawbacks. Besides only being able to address 4 GB of memory and performing 32-bit arithmetic, compatibility mode limits a process to 8 integer and 8 SSE registers, the calling convention passes parameters through memory, and only x87 floating point is available [19]. Position independent code (PIC), used heavily in libraries, can slow down libraries by 20% or more [16].

However, many workloads impacted by meltdown stand to benefit greatly from Shrink. This is because these workloads enter the kernel frequently and perform little computation. Running in 32-bit does not significantly affect their performance, yet PTI degrades their performance by increasing the cost of the system calls and interrupts. Many of these workloads may use less than <4GB of memory. For example, many AWS instances today fall under this category (t2.nano to t2.medium) [1]. On some platforms (e.g., Windows) binary distribution is common and many 32-bit binaries exist. For instance, Microsoft recommends that users run 32-bit Office [26]. In these cases, Shrink can nearly restore performance without penalty.

Implementation. We implemented Shrink on Linux 4.15-RC8 and submitted patches as a RFC to the Linux community [3]. Shrink is less than 300 lines of code, including comments. We support Shrink on a per address space (`mm_struct`) basis. When a 32-bit executable is loaded through `execve(2)`, we mark that address space as "shrunk" and disable PTI. Upon return to user mode, the process runs in 32-bit compatibility mode and is unable to address any kernel mappings, which exist outside of the 32-bit address space. When a kernel transition occurs, we do not update CR3 if the process is shrunk. We allow global mappings, so kernel mappings persist across system call invocations and interrupts. However, if a non-shrunk process is scheduled, we disable global mappings by clearing the appropriate processor control register (CR4.PGE). We protect against speculative execution of userspace pages from the kernel by using SMEP on CPUs that support it. One benefit of PTI we do not provide is SMEP emulation on unsupported processors. However, SMEP is regarded as a hardening mechanism rather than a defense against a specific attack [32].

A 32-bit process can switch into 64-bit mode by loading the 64-bit user code segment descriptor. Shrink protects against this by marking the 64-bit user code segment descriptor as not present. If a user loads a 64-bit code segment, a non-present (`#NP`) exception is delivered to the kernel, which we handle by re-enabling PTI. Although rare, some 32-bit applications, such as Checkpoint/Restore In Userspace (CRIU) load 64-bit code segments [10].

4 EVALUATION

Our testbed consists of machines from four Intel x86 microarchitectures of the last decade: Nehalem (Xeon E6540), Ivy Bridge (Xeon E5-2650 v2), Haswell (Xeon E5-1620 v3) and Skylake (Xeon Gold 5120).

To manage the 32-bit userspace, we use docker containers with 2 CPUs each. We run tests in one of three configurations under Linux 4.15-RC8: *unsafe*, a meltdown-vulnerable configuration using the `pti=off` boot option, *pti*, which runs the default PTI option, and *shrink*, which runs with our Shrink patches applied. We run each configuration on bare metal

(native) and in a VM (virtual), where we used KVM with 4 CPUs allocated per VM.

Each graph depicts performance normalized to pti=1. Raw performance numbers appear on the top of each bar, and the error bars represent standard deviation (expressed as percent of average).

4.1 Shrink security

We tested the Meltdown proof of concept [23] with shrink and verified that it was not able to read kernel memory. There is no exploit that enables 64-bit addresses to be used in 32-bit mode which we are aware of. Intel engineers responded to our RFC [3] but did not raise concerns regarding the security of our approach.

4.2 Microbenchmarks

We microbenchmarked each configuration using `lmbench` [24] 3.0 and show the performance of system calls in Figure 2. PTI results in a performance drops across all microarchitectures: 4–8× in native setups. In virtualized setups, the degradation is as much as 11×. Shrink nearly restores the performance of system calls with only a ~10ns overhead that can be attributed to the more expensive system call convention in 32-bit mode. An overhead breakdown indicates that ~90% of it is due to switching address space and all the rest is due to TLB misses.

4.3 Workloads

We evaluated several common I/O-bound workloads using shrink: iPerf, Nginix and Redis.

iPerf. iPerf is a popular network measurement tool for Linux. We test the performance of sending small (10 byte) messages over loopback using TCP in in Figure 3. On native, the overhead of PTI is between 1.5–2×, roughly decreasing with each new microarchitecture. In virtual, the overhead is 2.2–3×. In all cases, shrink is able to nearly match the performance of the unsafe configuration.

Nginix. Nginix 1.10.3 is a widely deployed web server. We used `apachebench` 2.3 to benchmark the performance of servicing a 10KB file. On native, the overhead of PTI ranges between 1.1–1.3×. Shrink is able to restore some of the performance for older microarchitectures, such as Nehalem, but on newer architectures, shrink does not significantly improve performance. This could be due to a variety of microarchitectual details. In virtual environments, shrink is able to restore some, but not all of the 1.2–1.6× overhead of PTI.

Redis. Redis 3.2.6 is an in-memory store used in many applications as a database or cache. We use the popular `redis-benchmark` utility and report the performance of `GET` (Figure 5) and `LRANGE 600` (Figure 6). Most other Redis

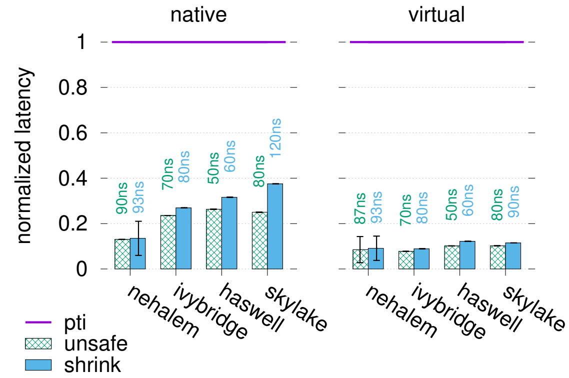

Figure 2: *Lmbench null call.*

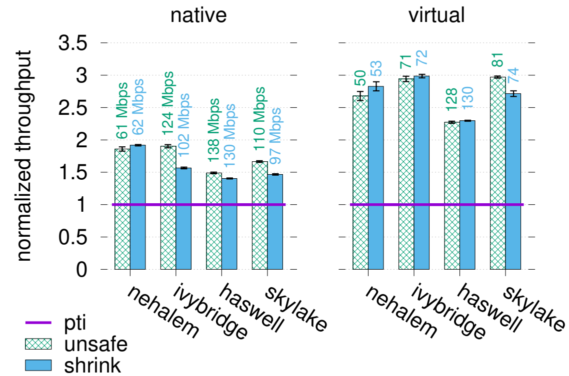

Figure 3: *Iperf.*

benchmarks performed similarly to `GET`, with overheads of 1.1–1.6×, and shrink restoring unsafe performance. But on two sets of benchmarks (`LRANGE`, `MSET`), shrink reduced performance by about 1.25×, while PTI added negligible overhead. We suspect that this could be due to the reduced number of registers available in 32-bit mode for traversing and copying elements out of the linked list. This result shows that shrink is not a panacea: we must test and profile our workloads, even if they fit in 4GB, to obtain maximum performance.

5 OTHER DIRECTIONS

Memory segmentation. Memory segmentation is a feature of the x86 architecture which provides an alternative to paging for memory protection. Unfortunately, the x86-64 bit architecture no longer supports segmentation in 64-bit mode. However, since many 32-bit processors are affected by meltdown, and those processors will suffer the most from PTI due to the lack of advanced features, segmentation may present itself as a reasonable alternative to PTI. Intel still manufactures 32-bit processors (e.g, Atom E680T) [18] used by many embedded systems [7, 17, 29, 30].

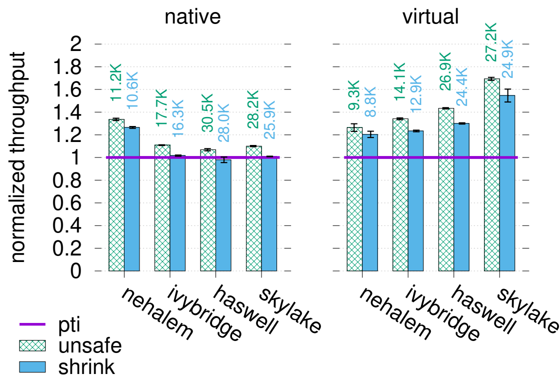

Figure 4: *Nginx.*

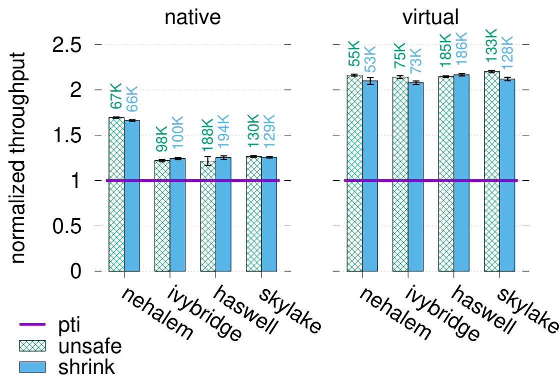

Figure 5: *Redis Get.*

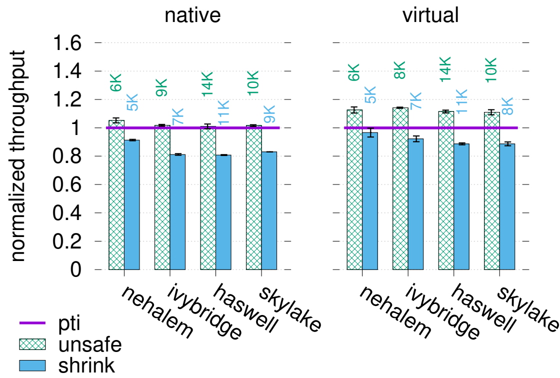

Figure 6: *Redis Lrange-600.*

We observe that it is sufficient to set the segment limit to exclude kernel addresses. We have implemented a proof of concept on Linux 4.15-RC8 and submitted a RFC [4] to the Linux kernel community. Notably, Linux Torvads replied that the Linux kernel used to use segmentation for protection, but the patch may not be reliable for all architectures. We have tested that our patch prevents the meltdown proof of concept from working on our testbed, and our experiments have shown empirically that it should not be possible to load data speculatively which is outside the segment limit.

Compilation-time safety checks. Another approach is to check programs during compilation by having the compiler ensure no kernel memory is referenced by masking pointers, and by employing control flow integrity techniques to prevent speculative execution of malicious code injected after compilation. The executable could be signed after compilation, which would allow the kernel to disable PTI for these safe executables.

Dynamic binary translation. Dynamic binary translation can be used to prevent binaries from referencing kernel memory, but doing so efficiently and securely is admittedly challenging, as potentially every memory access needs to be masked.

6 POSTSCRIPT

We submitted patches implementing Shrink to the Linux kernel, and after a lengthy review process, our patches were ultimately rejected by Linus Torvalds, who accepted that the technique would work, but could transparently fail, and that it would be better to accept the overhead of PTI [5]. While some time has passed since our evaluation and submission, the PTI mechanism has remained essentially the same. However, on systems without PCID support, the overhead of PTI was deemed to be too high, so the kernel developers ultimately decided to sacrifice some security by mapping the kernel "text" section (code) as global to reduce syscall runtime [14]. This exposes the kernel code to speculative execution attacks which renders certain security hardening mechanisms, such as KASLR and C structure randomization ineffective, and is automatically disabled on certain configurations [13]. Shrink can offer an alternative which does not result in the same exposure to side channel attacks.

PTI overhead depends on many factors. In virtualization, using huge-pages to back the VM memory may reduce the performance overhead, especially when PCID is not supported by the CPU by reducing the number of TLB misses. Using multiple `seccomp` security profiles as done by systemd can increase the number of TLB misses, increasing overhead.

In addition to protection against Meltdown, Shrink provides several additional benefits. First, it prevents the leakage of kernel addresses through speculative page-walks that might circumvent Kernel Address Space Layout Randomization (KASLR) [11]. Second, it reduces the overhead of TLB invalidations, which are caused by `munmap` syscall or memory migration, as unlike PTI, "shrink" requires only one page-table to be flushed. Note that the workloads that we ran did not cause such frequent TLB invalidations, which might induce

even greater overhead. Finally, PTI introduces some (although small) memory overheads, which “shrink” eliminates.

Years after its discovery, Meltdown still affects billions of processors deployed and sold even today. The semiconductor shortage [33] means that vulnerable processors from older process nodes, without PCID support [2] are likely to remain in the supply chain for years to come. While our patches were ultimately rejected by Linux, Shrink still holds value as a performant, secure alternative to the partial PTI employed by the kernel today.

REFERENCES

[1] EC2 instance types. https://aws.amazon.com/ec2/instance-types/. Accessed 25 Jan 2018.

[2] Amazon. Beelink u55 mini pc windows 10 pro, intel core i3-5005u. https://www.amazon.com/Beelink-Windows-i3-5005U-Processor-Ethernet/dp/B085RLQN5Z, 2021.

[3] Anonymous. Anonymized for blind review, January 1 2018. Accessed 25 Jan 2018.

[4] Anonymous. Anonymized for blind review, January 1 2018. Accessed 25 Jan 2018.

[5] Anonymous. Anonymized for blind review, January 1 2018.

[6] About speculative execution vulnerabilities in arm-based and intel cpus. https://support.apple.com/en-us/HT208394.

[7] AB3000 series - avionics i/o computers. https://www.astronics.com/ballard-technology/small-form-factor-systems/ab3000-series-avionics-i-o-computers. Accessed 25 Jan 2018.

[8] CERT. Vulnerability note VU#584653. https://www.kb.cert.org/vuls/id/584653. Accessed 25 Jan 2018.

[9] Intel Corporation. Affected processors: Transient execution attacks & related security issues by cpu. https://software.intel.com/security-software-guidance/processors-affected-transient-execution-attack-mitigation-product-cpu-model, 2021.

[10] 32bit tasks c/r. https://criu.org/32bit_tasks_C/R. Accessed 25 Jan 2018.

[11] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is dead: long live KASLR. In *International Symposium on Engineering Secure Software and Systems*, pages 161–176. Springer, 2017.

[12] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 368–379. ACM, 2016.

[13] Dave Hansen. x86, pti: disallow global kernel text with RANDSTRUCT. https://lkml.org/lkml/2018/4/20/915, 2018.

[14] Dave Hansen. x86/pti: leave kernel text global for !PCID. https://lkml.org/lkml/2018/3/23/1081, 2018.

[15] David Hansen. Patchwork [v4] x86/doc: add PTI description. https://patchwork.kernel.org/patch/10146855/. Accessed 25 Jan 2018.

[16] Milind Girkar H.J. Lu, H Peter Anvin. x32 - a native 32bit ABI for x86-64. "http://linuxplumbersconf.net/2011/ocw/system/presentations/531/original/x32-LPC-2011-0906.pptx". Accessed 25 Jan 2018.

[17] CompactPCI-PlusIO-SBC F75P from MEN Mikro Elektronik GmbH. https://www.infoteam.de/en/industry/railway/safe-train-control/. Accessed 25 Jan 2018.

[18] Atom processor e680. "https://ark.intel.com/products/52497/Intel-Atom-Processor-E680-512K-Cache-1_60-GHz". Accessed 25 Jan 2018.

[19] Intel® 64 and IA-32 Architectures Developer Reference Manual Volumes1-4, December 2017.

[20] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 380–392. ACM, 2016.

[21] Greg Kroah-Hartman. Linux 4.14.11 changelog. https://cdn.kernel.org/pub/linux/kernel/v4.x/ChangeLog-4.14.11. Accessed 25 Jan 2018.

[22] Michael Larabel. KPTI + retpoline linux benchmarking on older clarksfield / penryn thinkpads. https://www.phoronix.com/scan.php?page=article&item=pre-pcid-kptiretpoline&num=6. Accessed 25 Jan 2018.

[23] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018.

[24] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

[25] Adv180002 | guidance to mitigate speculative execution side-channel vulnerabilities. https://portal.msrc.microsoft.com/en-us/security-guidance/advisory/ADV180002. Accessed 25 Jan 2018.

[26] Choose between the 64-bit or 32-bit version of office. "https://support.office.com/en-us/article/choose-between-the-64-bit-or-32-bit-version-of-office-2dee7807-8f95-4d0c-b5fe-6c6f49b8d261". Accessed 25 Jan 2018.

[27] CVE-2017-5754. Available from NVD, CVE-ID CVE-2017-5754, https://nvd.nist.gov/vuln/detail/CVE-2017-5754, January 1 2018. Accessed 25 Jan 2018.

[28] Jeffery E Payne. Regulation and information security: can Y2K lessons help us? *IEEE Security & Privacy*, 2(2):58–61, 2004.

[29] Atom "tunnel creek" based EPU. http://www.sarsen.net/products/boards/single-board-computers/falcon/140. Accessed 25 Jan 2018.

[30] Siemens launches product line of particularly compact industrial pcs with new intel atom processors. https://www.siemens.com/press/en/pressrelease/?press=/en/pressrelease/2011/industry_automation/iia2011022615.htm. Accessed 25 Jan 2018.

[31] Gordon Tetlow. Response to meltdown and spectre. https://lists.freebsd.org/pipermail/freebsd-security/2018-January/009719.html.

[32] Linus Torvalds. x86/enter: Create macros to restrict/unrestrict indirect branch speculation. "https://lkml.org/lkml/2018/1/26/520". Accessed 28 Jan 2018.

[33] The Verge. Biden administration global semiconductor shortage executive order. https://www.theverge.com/2021/2/11/22278431/biden-administration-global-semiconductor-chip-shortage-executive-order, 2021.