What Every Hacker Should Know About TLB Invalidation

December 2024
Hackers to Hackers Conference (H2HC)

Paweł Wieczorkiewicz
Open Source Security, Inc.

grsecurity

PAWEŁ WIECZORKIEWICZ

EMAIL: WIPAWEL@GRSECURITY.NET
TWITTER / X: @WIPAWEL

- Security Researcher at Open Source Security, Inc. (creators of grsecurity®)
- Low-level security research of system software and hardware
- Reverse engineering and binary analysis
- Kernel Test Framework (KTF) creator and maintainer https://github.com/KernelTestFramework/ktf

Outline

- **Theory**
  - What is TLB?
  - What is the purpose of TLB?
  - Paging-structure caches demystified
  - Basic whys and whens of TLB invalidation
  - Optimizations
    - Global Pages
    - PCID and VPID
  - Details of TLB invalidation
- **Practice**
  - An interesting case study of a PaX/grsecurity PRIVATE KSTACKS bug
  - A case of broken INVLPG instruction on new generations of Intel CPUs

What is the Translation-Lookaside Buffer (TLB)?

What is the Translation-Lookaside Buffer (TLB)?

- Virtual Memory (VM) and Memory Management Unit (MMU)
  - A simplified view:
    - Software operates within **many various** virtual address spaces
    - Hardware operates within a **single** physical address space
      - Virtual address != physical address
        - Translation is needed

What is the Translation-Lookaside Buffer (TLB)?

- Virtual Memory (VM) and Memory Management Unit (MMU)
    - A simplified view:
        - Software operates within **many various** virtual address spaces
        - Hardware operates within a **single** physical address space
            - Virtual address != physical address
                - Translation is needed
    - Paging and page tables
        - Implementation “detail” of Virtual Memory
        - Physical memory split into frames of standard sizes (4K, 2M, 1G)
        - Virtual memory address space split into pages of physical frames sizes
        - Physical frames **mapped** to virtual address spaces to back virtual memory pages
            - Extra metadata of virtual memory pages: `Present`, `Read/Write`, `User/Superuser`, ...

What is the Translation-Lookaside Buffer (TLB)?

- Example schematic of 4-level page tables

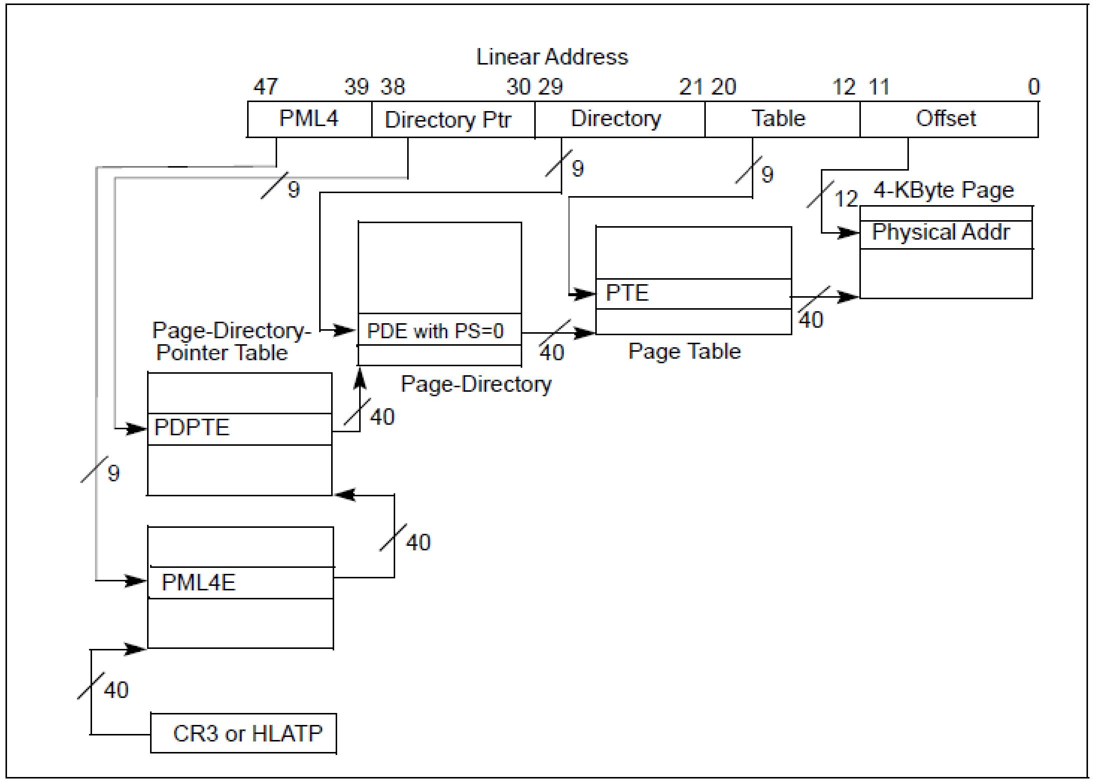
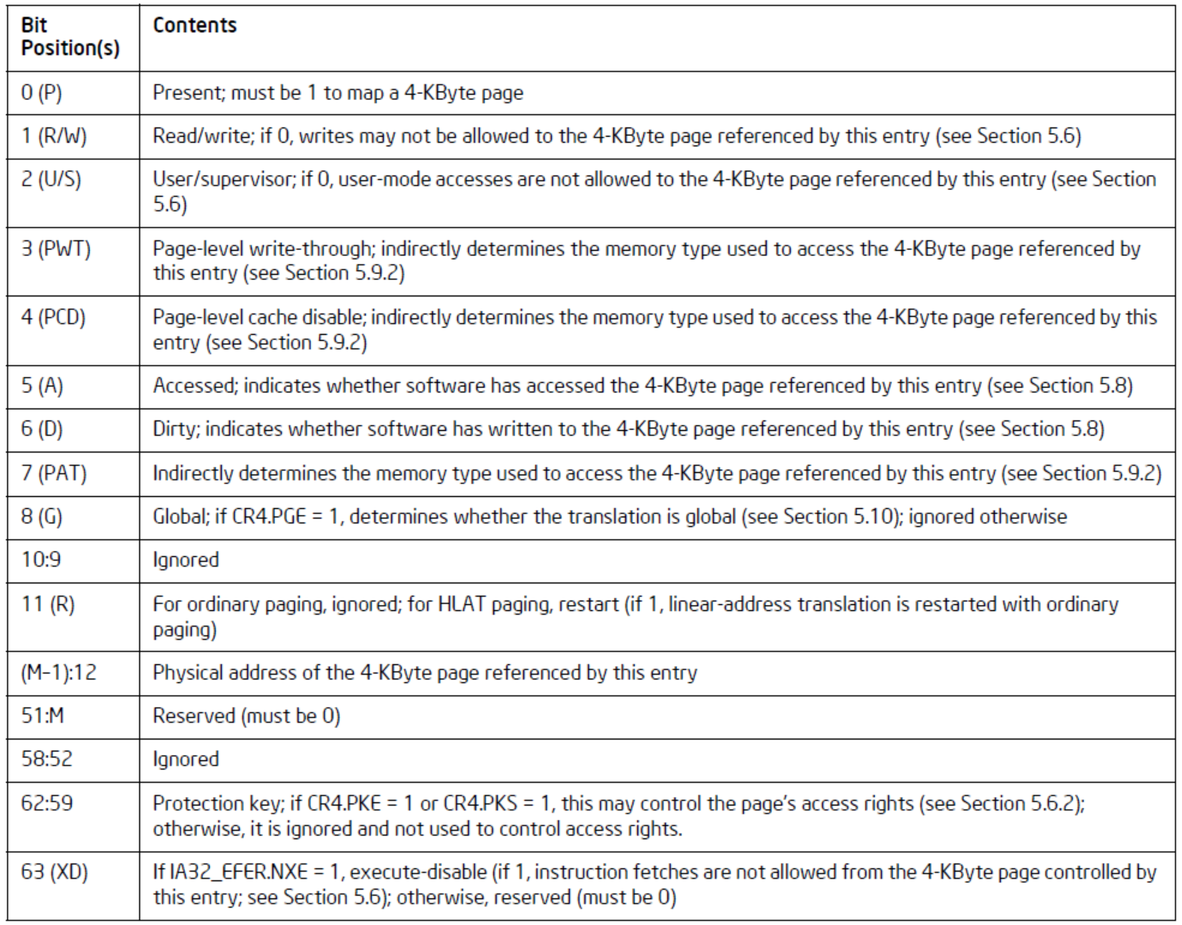


Figure 5-8. Linear-Address Translation to a 4-KByte Page Using 4-Level Paging

- Format of typical 4K virtual memory page

Table 5-20. Format of a Page-Table Entry that Maps a 4-KByte Page



Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 5.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 5.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 5.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 5.9.2)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 5.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 5.8)
7 (PAT)	Indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 5.9.2)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 5.10); ignored otherwise
10:9	Ignored
11 (R)	For ordinary paging, ignored; for HLAT paging, restart (if 1, linear-address translation is restarted with ordinary paging)
(M-1):12	Physical address of the 4-KByte page referenced by this entry
51:M	Reserved (must be 0)
58:52	Ignored
62:59	Protection key; if CR4.PKE = 1 or CR4.PKS = 1, this may control the page's access rights (see Section 5.6.2); otherwise, it is ignored and not used to control access rights.
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 4-KByte page controlled by this entry; see Section 5.6); otherwise, reserved (must be 0)

Source: Intel® 64 and IA-32 Architectures Software Developer's Manual

What is the Translation-Lookaside Buffer (TLB)?

- Virtual Memory (VM) and Memory Management Unit (MMU)
    - Paging and page tables – **page walk**
        - Every time a virtual address is being accessed by SW → CPU has to perform a page walk
            - CPU has dedicated page walk HW units to perform this process transparently
        - A page walk begins by getting physical address of a top page table from a dedicated register (`CR3` on x86)
            - This table memory has to be fetched, indexed, metadata flags need to be examined
        - From the fetched and indexed table next level page table physical address is obtained
            - Again, this table memory has to be fetched, indexed, metadata flags need to be examined
        - This traversing continues until the lowest level of the page table hierarchy provides actual memory address
        - **This process is very expensive**

What is the Translation-Lookaside Buffer (TLB)?

- TLB is a (set of) cache buffer(s) automatically storing previous, successful page walk results
- TLB caches **recent** virtual address to physical address **translations** …
    - … and a summary of traversed page table entries metadata flags

What is the Translation-Lookaside Buffer (TLB)?

- TLB is a (set of) cache buffer(s) automatically storing previous, successful page walk results
- TLB caches **recent** virtual address to physical address **translations** …
  - … and a summary of traversed page table entries metadata flags

- Typical TLB consists of entries referenced by a virtual page number (e.g. `virtual address >> 12`)
- Typical TLB entry consists of:
  - Physical frame number (physical address = (`frame number << 12`) + `offset`)
  - Access rights from page tables' entries used for the translation:
    - Logical AND of Read/Write flags
    - Logical AND of User/Superuser flags
    - Additional page table metadata attributes

What is the Translation-Lookaside Buffer (TLB)?

- Organization of TLB hardware may differ across various CPU microarchitectures
  - TLB can be split into independent: **instruction** TLB (iTLB) and **data** TLB (dTLB)
    - Different number of entries
      - Due to different spatial and temporal characteristics of accesses
    - Different metadata flags cached
      - For example: code does not need a dirty flag

  - Multiple levels of TLB buffers
    - For example:
      - First level: iTLB + dTLB
      - Second level: unified TLB

What is the purpose of TLB?

- Probably obvious at that point → **PERFORMANCE**
- TLB caches recent translations obtained with **expensive** page walks
  - A principle of temporal locality applies:
    - Recently used virtual memory page is likely to be re-used again soon
  - Each TLB Hit is a huge win
- TLB acting as an intermediate cache allows to:
  - Pipeline and/or parallelize **fast** TLB Hits with **slow** TLB Misses
    - Outsource expensive and slow work to hardware page walk units …
    - … while keep serving TLB Hits quickly

TLB quirks and peculiarities

TLB quirks and peculiarities

- TLB caches entries only for **successful** page walks
    - Each page table level entry used for a translation has to have:
        - **Present** flag set
        - **Reserved** bits zeroed
    - Intel SDM:
        - *“the processor does not cache a translation for a page number unless the accessed flag is 1 in each of the paging-structure entries used during translation…”*
            - Software can use it for deciding if invalidation is needed
        - *“… before caching a translation, the processor sets any of these accessed flags that is not already 1”*
            - This can be used for page walk detections

    - This obviously affects the need for invalidation

TLB quirks and peculiarities

- Intel SDM: "*the processor may cache a translation for any linear address, even if that address is not used to access memory*"
    - TLB may store for example:
        - Translations for memory **prefetches**
        - ***"accesses that result from speculative execution that would never actually occur in the executed code path"***
            - Side-channels, side-channels everywhere!

- Intel SDM: "*On a processor supporting Hyper-Threading Technology, invalidations performed on* ***one*** *logical processor may invalidate entries in the TLBs and paging-structure caches used by* ***other*** *logical processors*"

- In other words: making assumptions about presence or absence of TLB entries is **risky**!

TLB quirks and peculiarities

- TLB may cache a huge page (e.g. 2M or 1G) translation as multiple 4K pages translations
    - This implementation detail is not visible to software
        - It may complicate explicit invalidation requirements for such huge page translation
- TLB may contain several distinct translations for the same physical memory addresses at once
    - For example:
        - TLB contains an existing 4K virtual page translation
        - Software modifies a corresponding page table entry, so the virtual page size changes (e.g. 2M)
        - Accessing the virtual page now may lead to allocating a completely new TLB entry
    - Which entry should be used for subsequent accesses?
        - Intel SDM: "*Which translation is used may vary from one execution to another, and the choice may be implementation-specific*"
            - Making assumptions is **risky** again!

An introduction to paging-structure caches

Introduction to paging-structure caches

- TLB is **not** the only cache buffer used for optimizing expensive page walks
- Modern CPUs usually implement also caches for higher-level page table entries
    - In addition to the TLB's:
        - A memory address final translation
            - Obtained from the lowest-level page table entry
        - A summary of all-level page table entries metadata flags

Introduction to paging-structure caches

- The principle of temporal and spatial locality applies again:
  - Regardless of a TLB capacity and its state, the same higher-level page table entries may be re-used frequently
    - Because they participate in mapping larger portions of virtual address spaces…
    - … or may be used by cross-address-space mappings of common, shared data (e.g. kernel code, libraries)
- Caching them may noticeably improve performance of many workloads on modern OSes
- Typically distinct paging-structure caches exists for all higher-level page tables
- Caching higher-level page table entries help reduce a page walk cost → **performance**
  - Any cache hit **shortens the page walk**
    - The lower the level → the shorter the page walk necessary

Introduction to paging-structure caches

- Exemplary PDPTE (3rd level page table) cache properties
  - Entries indexed with only a part of the virtual address – depending on the paging mode
    - Example: 4-level paging mode → bits **47:30**, just 18 bits
  - Typical cache size is **small** – depends on a CPU microarchitecture implementation
  - Entries contain:
    - Physical address of a PDE page table (2nd level page table)
    - A summary of this and higher-level page tables entries metadata flags
      - Logical AND for: `Read/Write` and `User/Superuser`
      - … etc

- Similar characteristics and properties apply to all other (higher and lower level) paging-structure caches

Paging-structure caches – more details

- Huge pages entries are not cached in paging-structure caches
    - 1G pages have `PS` bit set in the 3rd level page table entries
        - No need for lower-level page table entries
    - 2M pages have `PS` bit set in the 2nd level page table entries
        - No need for PTE
    - These are handled by TLB

- Similarly to TLB requirements:
    - Only entries of present and valid (no reserved bits set) pages are cached
    - Accessed flags of the page table entries involved in the page walk are set by the CPU

Paging-structure caches – more details

- According to documentation: **CPU is free to create a paging-structure cache entry at any time**
  - Even when there are no translations for virtual addresses using the entry
  - Similar to the TLB case:
    - *“The processor may create entries in paging-structure caches for translations required for prefetches and or accesses that are a result of speculative execution that would never actually occur in the executed code path”*

- Already created cache entry may remain unmodified regardless of software’s subsequent modifications to the page tables
  - **Need to explicitly invalidate**

- Entries are associated with the current PCID – more on this later

TLB and paging-structure caches interaction

- How does CPU use TLB with paging-structure caches?
    - In case of a direct **TLB hit** → TLB entry data is used immediately
    - In case of a **TLB miss**:
        - Paging-structure cache for 2nd level page table entries (PDE) may be consulted
            - In case of a **cache hit** → quick page walk (just PTE is needed)
            - In case of a **cache miss**:
        - Paging-structure cache for 3rd level page table entries (PDPTE) may be consulted
            - In case of a **cache hit** → shorter page walk (just PDE and PTE needed)
            - In case of a **cache miss**: …
        - … and so on depending on the caches available…

    - If CPU misses in all the caches → a full page walk is scheduled

Why and when a TLB invalidation is needed?

TLB invalidation – why?

- In case of a **TLB hit**:
  - CPU may use found TLB entry to determine a **page frame**, **access rights**, and other attributes
  - CPU will likely **NOT** consult the page tables in memory
- As mentioned earlier:
  - “*The processor may retain a TLB entry unmodified even if software subsequently modifies the relevant paging-structure entries in memory*”
    - … and same is true for paging-structure caches!
- In other words:
  - Invalidation is needed to maintain coherency between translation caches (TLB, paging-structure caches) and page tables in memory **at all times**
    - Without this coherency: system stability and security is gone

TLB invalidation – when?

- Whenever the in-memory page table entries and their potentially cached values might be out-of-sync?
  - Excellent guarantees, but terrible performance
- Whenever software modifies page tables?
  - Not necessarily, remember the requirements for caching (Present bit set, etc)
    - Some modifications can assume the entry is not in cache
  - Whenever virtual address space changes?
- Whenever one process (page table A) is context switched into another process (page table B)?
  - What if they share some memory pages?
  - What about common kernel code?

TLB invalidation – when?

- Invalidating too much is very bad for performance
  - Operating systems try to invalidate only:
    - When necessary and …
    - … what really needs to be flushed

- Invalidating too little however makes performance not matter at all

- In real-life situations discussed later, we will see how problematic this can be
  - Some optimization assumptions might not hold …
    - … and be super complex to reason about
  - And… of course… a broken CPU microcode...

Hardware optimizations

Hardware optimizations

- Hardware features have been added to CPUs in order to:
    - Help software maintain the translation cache coherency more efficiently
    - Eliminate unnecessary invalidation of frequently used translation entries
- The features:
    - **Global Pages**
    - Process Context Identifiers (**PCID**)
    - Virtual Processor Identifier (**VPID**) – virtualization feature (vCPU tagging)
- All the features are optional, but can be also used all together
    - Most if not all of modern OSes use PCID
    - Some of the OSes additionally rely on Global Pages
        - Examples: FreeBSD

Hardware optimizations – Global Pages

- Global Pages feature was added to limit unnecessary invalidations of frequently used pages
  - For example: shared memory pages between different virtual address spaces
    - Common code (libraries)
    - Always mapped-in kernel memory

- Whenever virtual address space is being changed (on a task switch or process context switch)
  - The corresponding page tables have to be changed as well
    - Thereby, on x86 a full TLB flush is triggered by:
      - All writes to the CR3 register
      - A hardware task switch

- Flushing and then hitting TLB misses for the same memory accesses is a waste

Hardware optimizations – Global Pages

- Enabling Global Pages feature (`PGE` bit of `CR4` on x86) makes CPU respect addition flag in page table entries
    - Global flag (G)

- Software can decide what are the shared or frequently used virtual memory pages and mark them as global
    - By setting the G bit in their lowest-level page table entry
    - Note: there is no G bit for higher-level page table entries

- When enabled, TLB entries with a global flag are not invalidated automatically when page tables are reloaded
    - Automatic avoidance of TLB misses for shared memory

Hardware optimizations – Global Pages

- However…
  - Global pages take up TLB entries
    - A good balance is needed to avoid TLB thrashing
  - Invalidation of global pages is more complicated
    - Might not be worth it, when it has to occur often

- Some quirks:
  - Global pages do not affect the behavior of paging-structure caches at all
    - Only higher-level page table entries cached there
  - Global pages entries in TLB might be used by CPU regardless of their specific PCID
    - Global means global after all… it’s getting complicated

Hardware optimizations – Process-Context Identifiers (PCID)

- PCID feature was added to limit **the need** for translations invalidation

- It enables software to tag different virtual address spaces with different (almost) unique identifiers
- … and thereby allows CPUs to easily distinguish between different virtual address spaces (tag matching)
  - CPU can consume cached translations only for the **current** address space (current PCID)
  - No need to invalidate other virtual address space translations proactively
  - Hardware will flush translation entries when it needs space for new ones (e.g. using LRU algorithm)

- Frequently used translations, even from different address spaces, can remain in the translation caches
  - Maintenance simplification
  - Big potential performance gain

Hardware optimizations – Process-Context Identifiers (PCID)

- PCID is a 12 bit identifier → up to 4096 different virtual address spaces can coexist
- PCID of zero is the default (also: current) one
    - Setting `PCIDE` bit in `CR4` register (on x86) enables non-zero PCIDs
- The current address space PCID is indicated by the 12 LSB bits of the `CR3` register (x86)
    - Switching page tables during process context switch allows to specify different PCID

- CPU creates entries in TLB **and** paging-structure caches tagged with the current PCID
    - … and uses **only** those entries that are tagged with the current PCID (unless global)
- **Hardware-level logical address space translations isolation**
- Disabling PCID (via unsetting `PCIDE` bit in `CR4`) triggers entire translation cache hierarchy invalidation
    - PCID of zero becomes again the current and only one PCID

TLB Invalidation details…

Operations triggering TLB invalidation

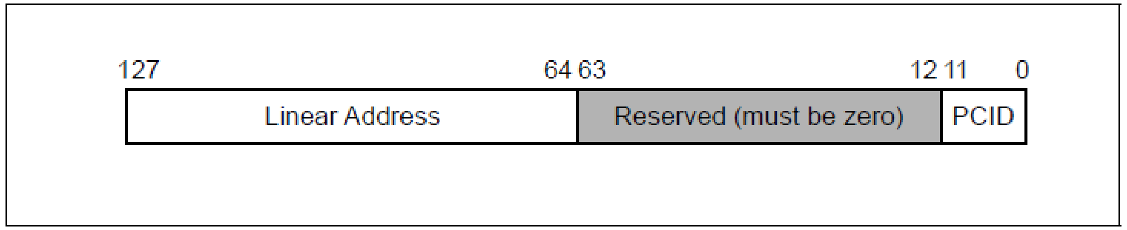


Figure 1-25. INVPCID Descriptor

- x86 instructions:
  - `INVLPG` *<virtual address>*
    - Invalidates any TLB entry for the specified virtual address **and** current PCID
    - Invalidates any global page TLB entry for the specified virtual address for **all** PCIDs
    - Invalidates **all** entries in **all** paging-structure caches for **all** virtual addresses of the **current** PCID
  - `INVPCID` *<type>*, *<descriptor>*
    - Descriptor points at 128 bit memory structure, where virtual address and PCID are specified
    - Type operand controls the scope of invalidation:
      - (0) **Address** – invalidates any TLB entry (not global) for the descriptor’s virtual address and PCID
      - (1) **Single context** – invalidates all TLB entries (not global) for the descriptor’s PCID
      - (2) **All contexts with globals** – invalidates all TLB entries for all virtual addresses and all PCIDs
      - (3) **All contexts** – invalidates all TLB entries (not global) for all virtual addresses and all PCIDs
    - Invalidates entries in **all** paging-structure caches according to the type and descriptor operands

Operations triggering TLB invalidation

- Other operations:
  - Disabling paging (e.g. unsetting `PG` bit of `CR0` on x86)
    - No surprises here – invalidates everything everywhere
  - Page table switch (e.g. writing to `CR3` on x86)
    - With PCID disabled – all TLB (but no globals) and paging-structure caches entries
    - With PCID enabled – depends on the MSB (63) bit of the `CR3` value
      - `0` – invalidates all TLB (no globals) and paging-structure caches entries for a new PCID
      - `1` – no invalidation (utilizing PCID feature isolation capabilities)
  - Disabling/Enabling of Global Pages feature
    - Invalidates everything everywhere
  - Disabling of the PCID feature
    - Invalidates everything everywhere

Operations triggering TLB invalidation

- Exceptions:
  - Page Fault exception (`#PF`) also invalidates entries in TLB and paging-structure caches
    - `#PF` exceptions result from accessing a virtual memory address
      - Various reasons: page not present, read-only, privilege violation, …
    - Upon `#PF` the following happens:
      - TLB entries for the `#PF` virtual address (stored in `CR2` on x86) **and** current PCID are invalidated
      - **Only** paging-structure caches entries relevant for the virtual address translation **and** current PCID are invalidated
    - `#PF` induced invalidations are to prevent reoccurring exceptions due to stale translation caches entries
- Virtualization transitions (`VMEnter` and `VMExit`) also invalidate TLB and paging-structure caches entries

TLB Shootdown – inter-processor invalidation

- TLB and paging-structure caches are typically per logical CPU resources - each logical CPU has their own
- On processors with multiple cores and/or supporting hyperthreading coordinated invalidations may be required
    - Upon modification of the in-memory page tables multiple CPUs may contain stale translation entries
    - Not enough for the page table modifying CPU to invalidate only its own translation caches
    - A TLB Shootdown operation must be initiated, so the translation cache hierarchy across all CPUs is in sync
- **TLB Shootdown**:
    - Usually implemented using execution barriers and inter-processor interrupts (IPI)
    - Procedure overview:
        - Initiating CPU before the in-memory page table modification traps all necessary CPU on a barrier
        - … modifies the page table and performs its own invalidation
        - … signals to the trapped CPUs to perform their invalidation
        - All CPUs resume execution

When software **has** to invalidate TLB?

- Modification of a page table entry mapping a virtual memory page
    - Software should issue `INVLPG` for all related virtual addresses
        - One `INVLPG` per page number is enough
- Modification of a page table entry mapping another page table entry
    - I.e. modification of a higher-level page table
    - Depending on the target page table's mappings count and their types:
        - Many mappings: full invalidation (e.g. via `CR3` register write) might be the best choice
        - Otherwise, individual `INVLPG` invalidations for the mappings might be optimal
        - Even if there is none – e.g. target page table entries have all pages not present
            - **Still at least one `INVLPG` is necessary!**
            - Why? You will see later…

When software **has** to invalidate TLB?

- Modification of a page table entry size
    - Huge page mapping
    - May lead to multiple translations in TLB for the same virtual addresses
        - One for the new huge page and one or more for the previous 4K pages
    - CPU may use **any** of the available translations
    - To avoid this, software has to first invalidate all affected translations

- Page table entry modification write might get re-ordered and get executed **after** a subsequent memory access
    - This can create a stale translation entry
    - Invalidation instructions are all **serializing** instructions

When software **has** to invalidate TLB?

- PCID identifier re-use
  - PCID value space is relatively small (12 bit – 4096 distinct values)
  - There may be more than 4096 virtual address spaces running concurrently on a system
  - Software has to re-use same PCID for different virtual address spaces
    - To avoid collisions in TLB and paging-structure caches:
      - Invalidate all translations for the new PCID
      - The `CR3` bit 63 serves that purpose, but does not invalidate global entries
      - For global entries more drastic measures have to be used:
        - All context with globals `INVPCID`
        - Disable → Enable Global Pages

When software **does not** have to invalidate?

- Modification of a page table entry mapping a non-present virtual page
  - Translations for non-present pages are not cached (neither in TLB nor paging-structure caches)
  - In other words: making a non-present page present does not require invalidation
    - Unless the page was present before and no invalidation has occurred since!
- Modifying access flag of a page table entry
  - `0 -> 1` – neither TLB nor paging-structure caches stores translations for untouched entries
  - `1 -> 0` – without invalidation CPU might not set the flag on a subsequent access – not fatal
- Modifying `Read/Write` flag of a page table entry
  - **Read-only → Writeable** – without invalidation `#PF` exception may occur
    - Not fatal, but sub-optimal
    - `#PF` will invalidate the cached translations
  - **Writeable → Read-only** – fatal without invalidation

When software **does not** have to invalidate?

- Modifying `User/Superuser` flag of a page table entry
    - **User → Superuser** – fatal without invalidation
    - **Superuser → User** - without invalidation `#PF` exception may occur
        - Not fatal, but sub-optimal
        - `#PF` will invalidate the cached translations
- Modifying dirty flag of a page table entry
    - `1 -> 0` – without invalidation CPU might not set the flag on a subsequent write – not fatal

When software does not have to invalidate **immediately**?

- Intel SDM quote: "*The processor may retain entries in the TLBs and paging-structure caches for an extended period of time. Software should not assume that the processor will not use entries associated with a linear address simply because time has passed*"

- **Example**: unmapping (or remapping) a part of a virtual address space
  - Unsetting page table entries `Present` flag
    - Two cases:
      - (case 1) Page table entry mapping a page
      - (case 2) Page table entry mapping another page table

When software does not have to invalidate **immediately**?

- What can go wrong without immediate invalidation in general here?
  - If OS re-uses the freed portion of the virtual address space
    - Stale translation entries point at different physical memory (or with different access rights)
      - Immediate fatal consequences
  - If OS re-uses the physical memory behind the freed portion of the virtual address space
    - Stale translation entries allow to access the same physical memory
      - Potentially with different access rights
    - “But nothing uses the freed virtual address space, so how bad can it be?”
      - Such assumptions usually die fast…
      - Speculative execution enters the scene…

TLB Invalidation – When Software does not have to invalidate **immediately**?

- What can go wrong without immediate invalidation in the case 2?
  - Intel SDM: "… *the processor may create an entry in a paging-structure cache even if there are no translations for any linear address that might use that entry*"
    - What does that mean?
      - CPU can populate paging-structure caches at its own discretion…

  - Also Intel SDM: "*if software has marked "not present" all entries in a page table, the processor may subsequently create a PDE-cache entry for the PDE that references that page table*"

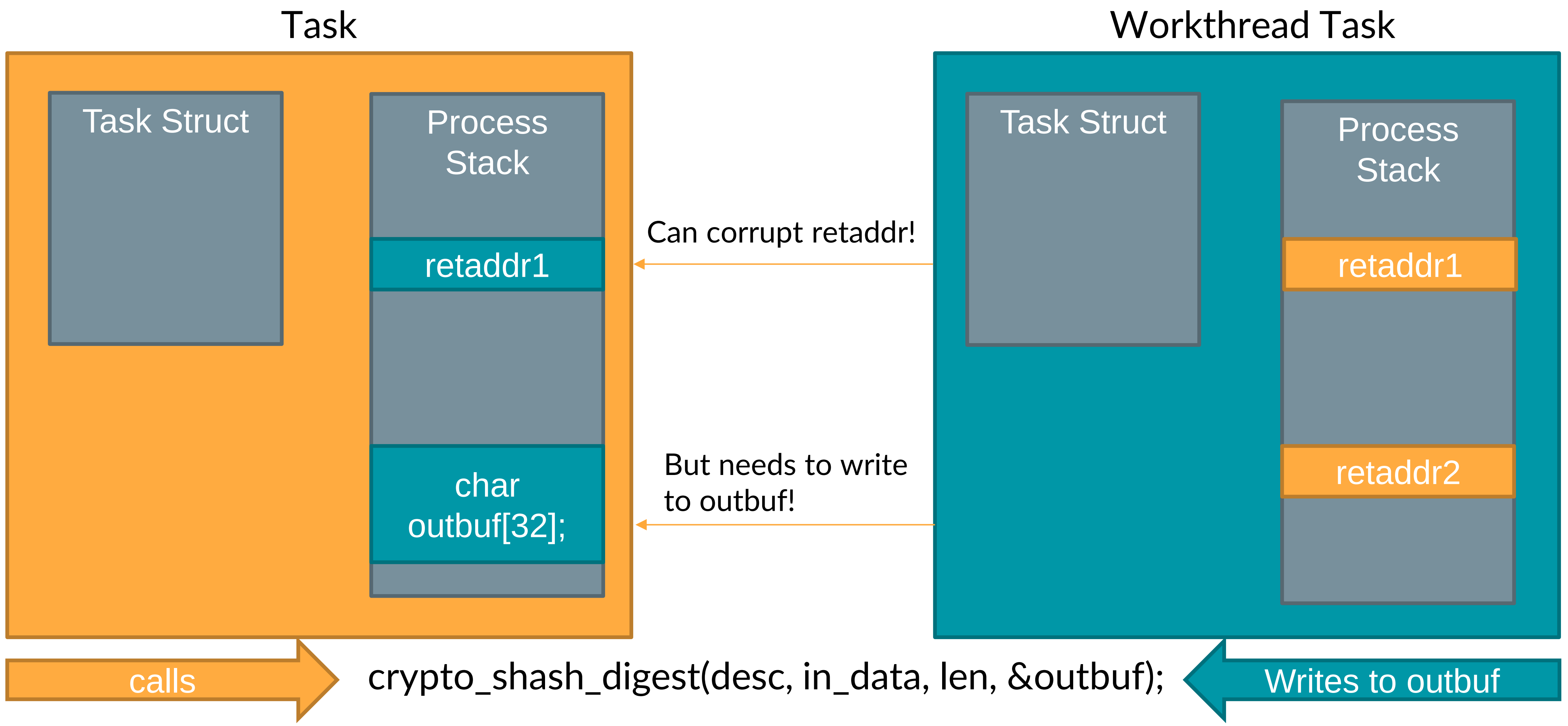
  - One may think that it does not sound too bad…
    - Well, let's see…

PaX/grsecurity PRIVATE KSTACKS bug

PaX/grsecurity PRIVATE KSTACKS

- What is the PRIVATE KSTACKS feature?
    - Private kernel stacks hardening enhancement introduced in PaX/grsecurity around February 2022
        - Kernel will disallow access to all process and IRQ kernel stacks except for the ones used by the currently running task on a given CPU
            - All-but-current process stacks not mapped
            - IRQ handlers cannot view other CPU IRQ stacks
        - This defence mechanism:
            - Completes the protection by PAX_RAP_RET and PAX_RAP_XOR
            - Prevents cross-kernel stack attacks and stack overflow/underflow attacks
            - Addresses attack that caused RFG to be shelved, without requiring CET/HW shadow stacks
                - Return Flow Guard - Microsoft attempt at backward-edge Control-Flow Integration
                - For shelved reason, see slide 27 of The Evolution of CFI Attacks and Defenses

PaX/grsecurity PRIVATE KSTACKS – Example (without)

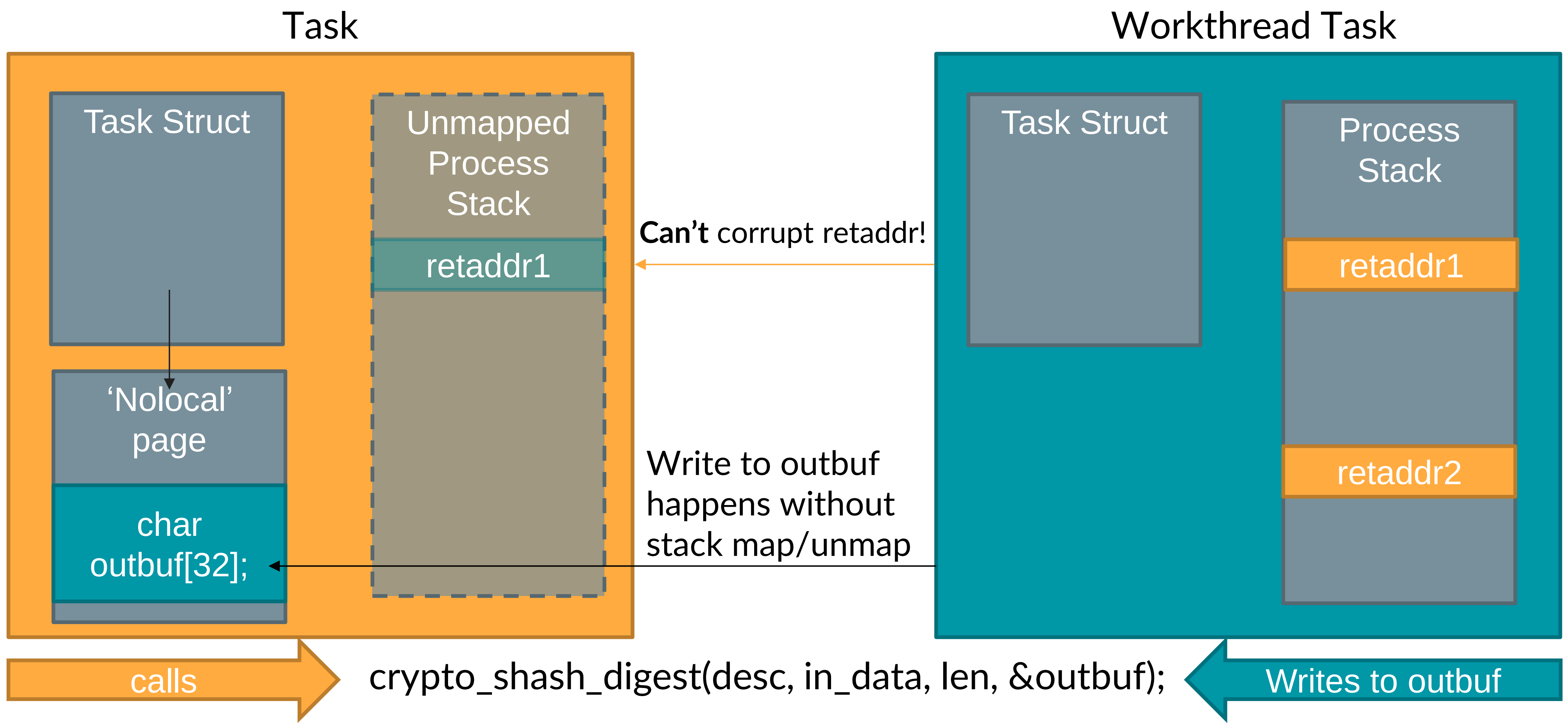


PaX/grsecurity PRIVATE KSTACKS – Example (without)

- We have two tasks of many on the system
    - A waiting task that just called a synchronous crypto API that will provide a hash digest in its on-stack output buffer
    - A running workthread task doing the actual work

- Without PRIVATE KSTACKS
    - The workthread task needs to write to a buffer on the waiting task's stack
    - But, this also allows bugs existing in that code to corrupt other parts of the other task's stack
        - Including its other local variables or return addresses

PaX/grsecurity PRIVATE KSTACKS – Example (with)



PaX/grsecurity PRIVATE KSTACKS – Example (with)

- With PRIVATE KSTACKS
  - The waiting task’s stack is completely unmapped from the perspective of the workthread task
    - The possibility to corrupt the return address or other local variables is gone
  - The workthread task still needs to write to the output buffer
    - A naïve approach would involve mapping/unmapping the other task’s stack fully
      - Not good for performance or security!
    - The PRIVATE KSTACKS feature changes out the backing storage for that stack variable
      - A new region for such “nolocal” variables allocation is pointed to by the associated task struct
      - This allows the output buffer to be visible to the workthread task
        - Without any performance-affecting map/unmap operations

PaX/grsecurity PRIVATE KSTACKS

- Contemporary systems usually consist of multiple CPU cores executing processes in parallel
- Due to a multitasking nature of OSes, these processes are constantly being scheduled and preempted
    - Usually there is more processes than CPU cores and all of them must receive CPU time
    - Process context switching involves page tables switching
        - … and usually requires some translations invalidation
    - Resumed processes might not continue execution on the same CPU
        - TLB Shootdown to keep translation caches coherent might be needed

PaX/grsecurity PRIVATE KSTACKS

- With PRIVATE KSTACKS feature all-but-current process stacks are not mapped
  - Upon process context switch the preempted process' stacks needs to be unmapped and a new process stack mapped
    - That involves page table entries modifications as well as TLB invalidation
  - Here, it also involves all kernel context stacks on and during every context switch
  - A meticulous selection of stacks allowed to be mapped is also needed
- A naïve implementation of this feature could involve a lot of page table operations leading to complex and expensive TLB maintenance requirements
- The actual PRIVATE KSTACKS feature implementation uses various techniques, that significantly reduce this complexity and cost → limiting the TLB invalidation requirements to the minimum
  - The most important one here: optimized page tables
    - Minimizing amount of mapping tables for each page table level
    - Page table re-use for kernel stacks mapping during context switching

PaX/grsecurity PRIVATE KSTACKS

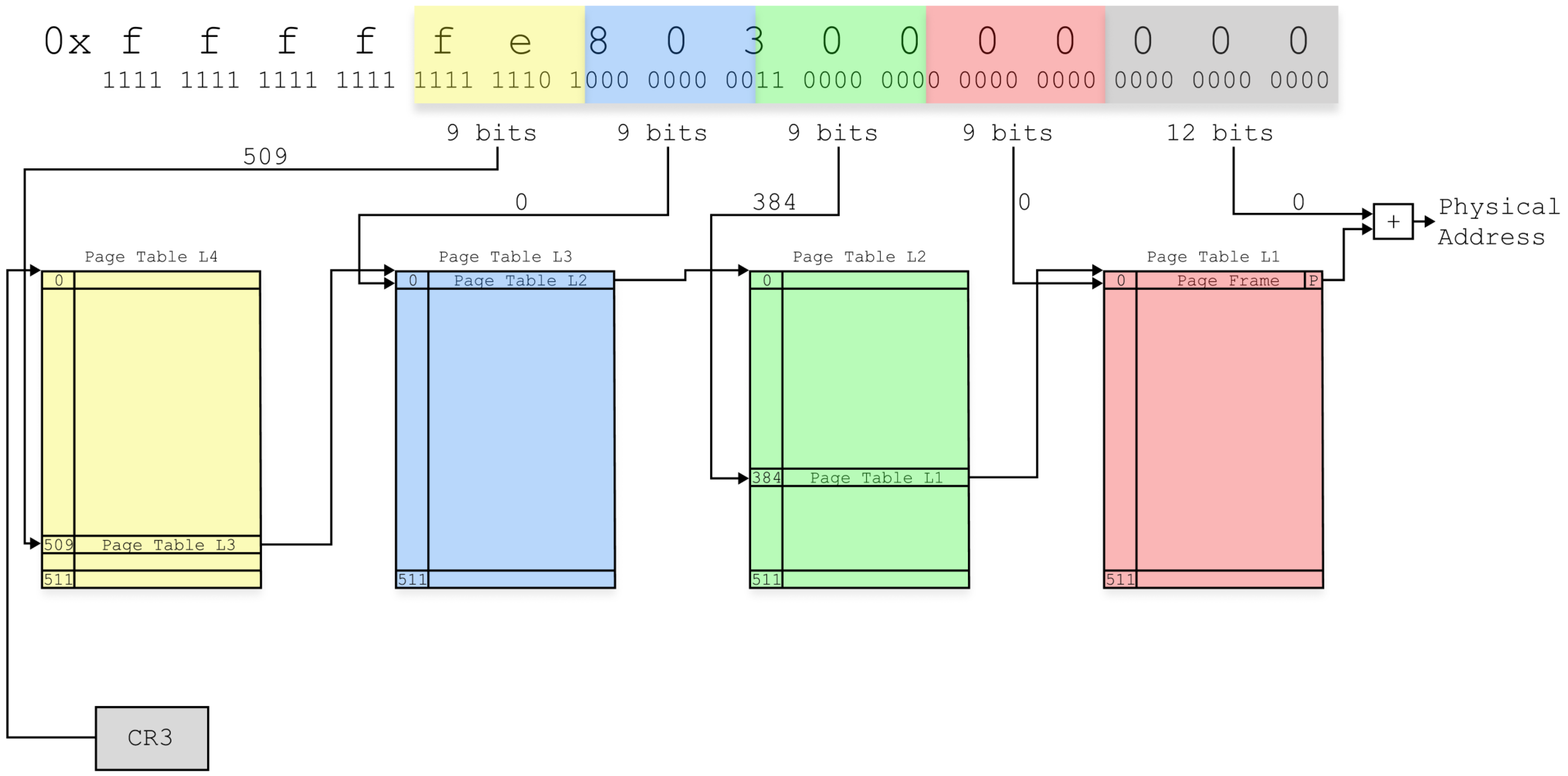
- Quite a while after the feature had been completed we started to observe weird things…
  - Rare, but reproducible `#PF` exceptions during process context switches
    - Always involving previous and next kernel stacks pages accesses
    - Impossible to explain based on preceding page tables operations and page tables content
    - Spurious in nature… just ignoring them was enough to restore proper execution
      - That suggested: no error in page tables, everything was mapped as expected
    - Never reproducible in virtualized environment (from within a VM)
    - Seemingly only reproducible on back-then new Intel CPUs of Alderlake family
      - These CPUs consists of so called P-cores (performance ones) and E-cores (slower ones)
      - The problem seemed to be bound to the E-cores only
  - Super rare (a handful of occurrences) RAP violations - indicating kernel stack data corruption
    - Impossible to explain and completely indeterministic

PaX/grsecurity PRIVATE KSTACKS

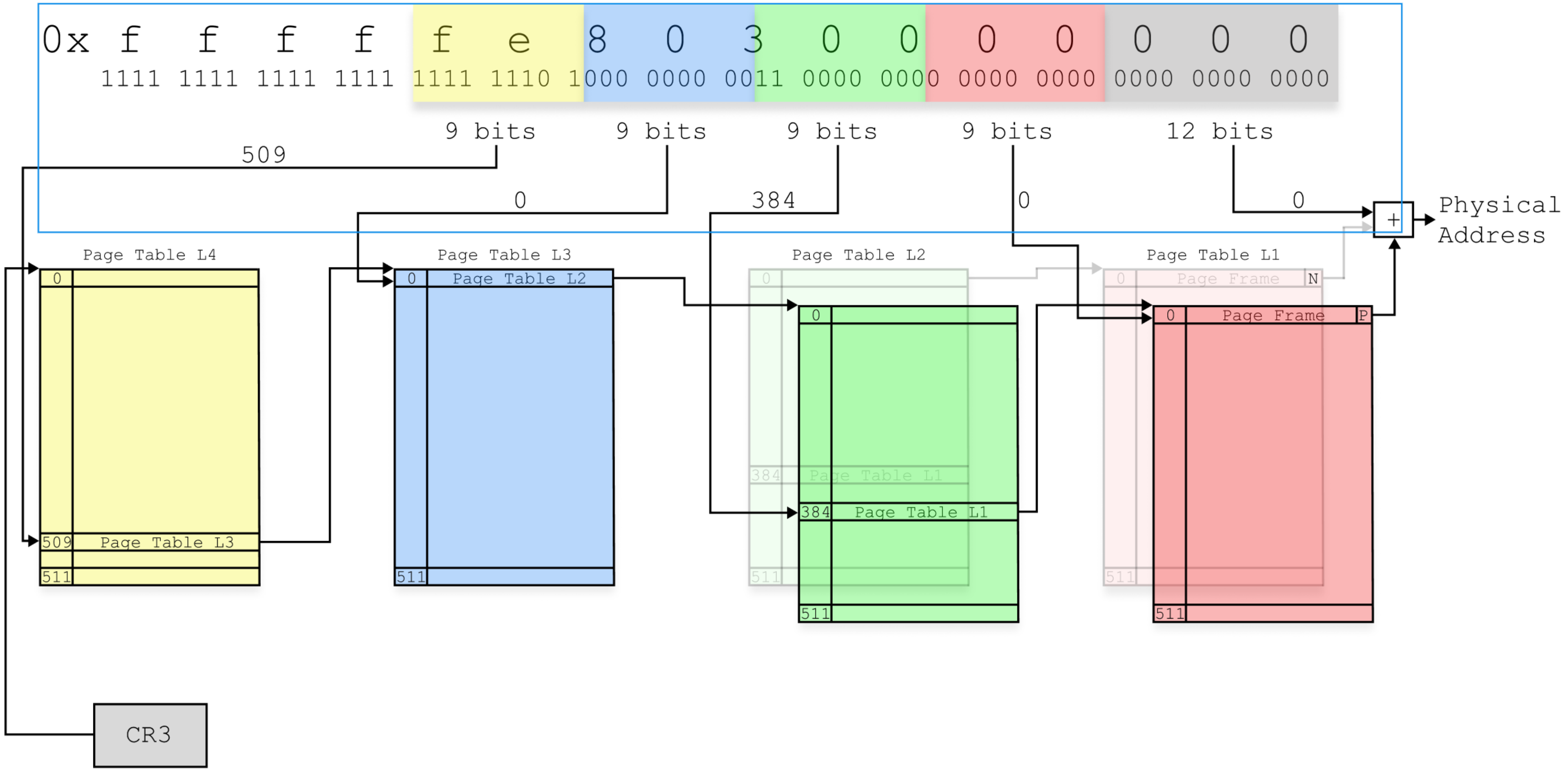
- How do you debug a bug that makes no sense?

- A lot of hours spent reproducing, limiting scope, narrowing down, tracing, documentation reading…
- Finally, at some point, we noticed a pattern
  - Always two re-used and consecutive 2nd level page tables were involved
    - New: currently mapped and previously unmapped
    - Old: currently unmapped and previously mapped
    - Both of the page table addresses had same bits 47:21
  - The `#PF` always involved a kernel stack address, whose page table entries indicated:
    - It is currently mapped using the new 2nd level page table entry
    - It **was** mapped just before when using the old 2nd level page table entry
    - It is not mapped (present) currently when using the old 2nd level page table entry

PaX/grsecurity PRIVATE KSTACKS – Page Tables (Before)



PaX/grsecurity PRIVATE KSTACKS – Page Tables (After)



PaX/grsecurity PRIVATE KSTACKS

- That rang a bell…
  - The paging-structure caches might be at play here…
  - Also, we weren't able to reproduce in a VM…
    - … and VMX transitions do flush the paging-structure caches

- But why?
  - The spurious nature of the `#PF` indicated that there was no mistake in page tables:
    - Previously unmapped page does not need invalidation as non-present pages translations are not cached
  - There was no architectural memory access that could potentially populate the translation caches with previously unmapped 2nd level page table
  - CPUs of other microarchitectures did not exhibit this problem at all

PaX/grsecurity PRIVATE KSTACKS

- Then a couple of documentation snippets caught our eye:
  - *"The use of the PDE cache depends on the paging mode: [...] For 4-level paging, each PDE-cache entry is referenced by a 27-bit value and is* ***used for linear addresses for which bits 47:21 have that value****"*
  - *"The processor may create an entry in a paging-structure cache even if there are no translations for any linear address that might use that entry. Thus,* ***if software has marked "not present" all entries in a page table, the processor may subsequently create a PDE-cache entry for the PDE that references that page table*** *(assuming that the PDE itself is marked "present")."*
  - *"The processor may create entries in paging-structure caches for translations required for prefetches and for accesses that are a result of speculative execution that would never actually occur in the executed code path."*
  - *"If the processor does not find a relevant TLB entry, it may use the upper bits of the linear address to select an entry from the PDE cache that is associated with the current PCID (Section 4.10.3.1 indicates which bits are used in each paging mode). It can then use that entry to complete the translation process (locating a PTE, etc.)"*

PaX/grsecurity PRIVATE KSTACKS

- That made us develop a hypothesis:
    - What if Alderlake CPUs' E-Cores are more "sensitive" and more likely to create and retain paging-structure caches entries…
    - … even for the no-translations-requiring page tables…
    - … merely based on a specific page table pages re-use and access patterns specific to the PRIVATE KSTACKS feature?

PaX/grsecurity PRIVATE KSTACKS

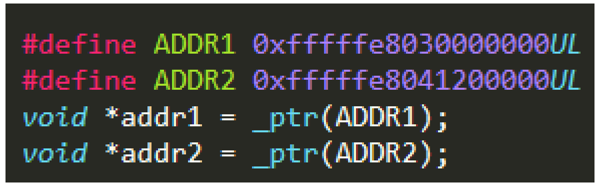
- More specifically, what if the following happens:
    - Previous kernel stack page is unmapped and its translations are invalidated
    - Current (new) kernel stack page becomes mapped, but does not have any entries in TLB (yet)
    - The previously used and now unmapped 2nd level page table's entry **remains** in paging-structure cache
        - It has the same indexing bits as the new page table page
    - When a new kernel stack access triggers a page walk, it uses stale entry from the paging-structure cache
        - When such misdirected page walk returns a non-present previous kernel stack's entry:
            - `#PF` exception is thrown
        - When such misdirected page walk returns a present kernel stack's entry:
            - A stack data corruption can happen…
                - … leading to the unexplainable RAP violations we used to observe

PaX/grsecurity PRIVATE KSTACKS – Proof-of-Concept

- Let’s test the hypothesis with a PoC…
- The PoC presented here is a simplified approximation of the chain of events, that happen during the PRIVATE KSTACKS context switch and that could lead to the unexpected `#PF` exceptions
- Primary objectives of the PoC were:
    - Prove paging-structure cache involvement and anticipated behavior
    - Prove the “uniqueness” of the Intel Alderlake CPUs
        - Same PoC should reproduce only of these CPUs
    - Reproduce the issue inside a VM, by better controlling of VMX transitions occurrences
    - Help to develop best (performance-wise) fix for the issue

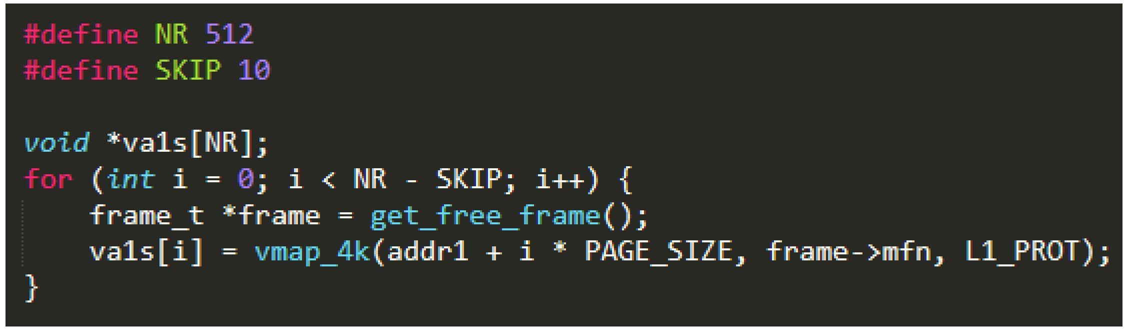
PaX/grsecurity PRIVATE KSTACKS – Proof-of-Concept

- The PoC begins by selecting two arbitrary virtual addresses to become the bases of two distinct address spaces



```c
#define ADDR1 0xfffffe8030000000UL
#define ADDR2 0xfffffe8041200000UL
void *addr1 = _ptr(ADDR1);
void *addr2 = _ptr(ADDR2);
```

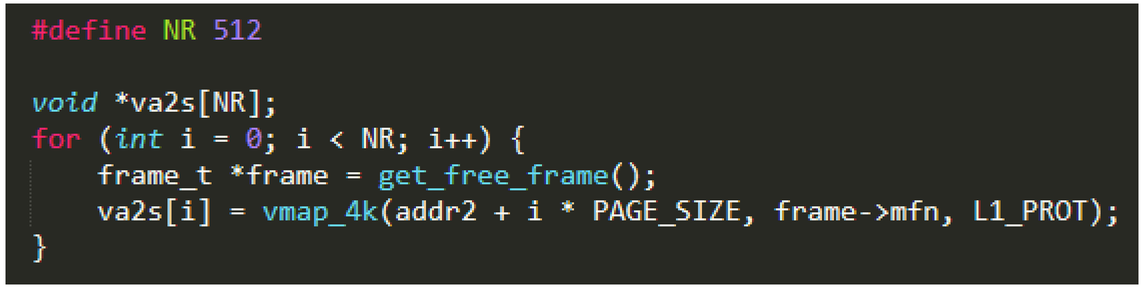
- Then, we allocate and map in the first address space, which consists of **502** consecutive 4K pages



```c
#define NR 512
#define SKIP 10

void *va1s[NR];
for (int i = 0; i < NR - SKIP; i++) {
    frame_t *frame = get_free_frame();
    va1s[i] = vmap_4k(addr1 + i * PAGE_SIZE, frame->mfn, L1_PROT);
}
```

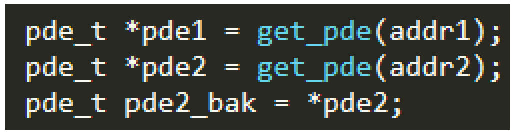
PaX/grsecurity PRIVATE KSTACKS – Proof-of-Concept

- Next, we allocate and map in the second address space, which consists of **512** consecutive 4K pages
    - A full 2MB range



```
#define NR 512

void *va2s[NR];
for (int i = 0; i < NR; i++) {
    frame_t *frame = get_free_frame();
    va2s[i] = vmap_4k(addr2 + i * PAGE_SIZE, frame->mfn, L1_PROT);
}
```

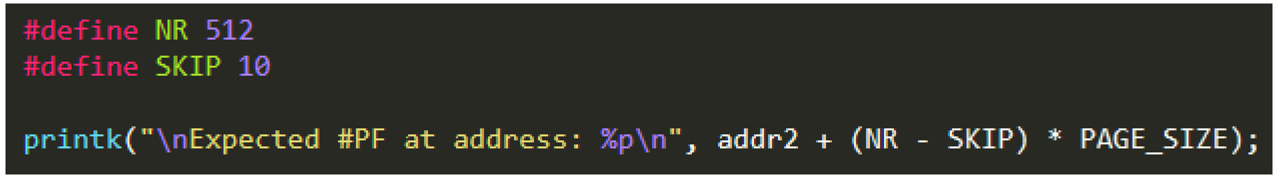
- Then, we obtain 2nd level page table entries for both of the base addresses and make a backup of the default content for the second address' entry



```
pde_t *pde1 = get_pde(addr1);
pde_t *pde2 = get_pde(addr2);
pde_t pde2_bak = *pde2;
```

PaX/grsecurity PRIVATE KSTACKS – Proof-of-Concept

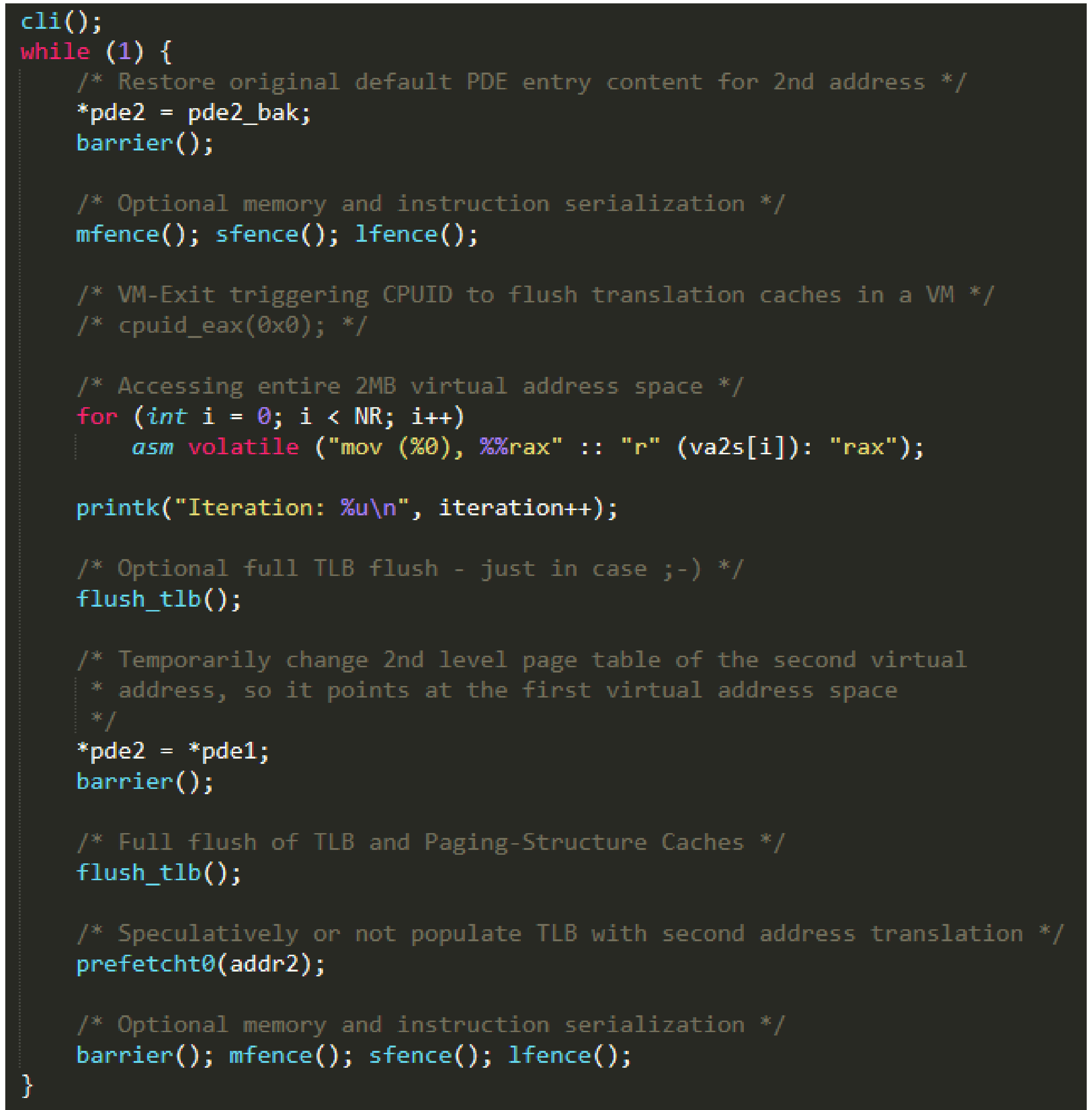
- Now, we can predict the virtual address of the expected spurious `#PF`
  - It is an address of the first unmapped page from the 2MB range of the first address space
    - Remember: we only mapped in 502 of 4K pages



```
#define NR 512
#define SKIP 10

printk("\nExpected #PF at address: %p\n", addr2 + (NR - SKIP) * PAGE_SIZE);
```

PaX/grsecurity PRIVATE KSTACKS – Proof-of-Concept

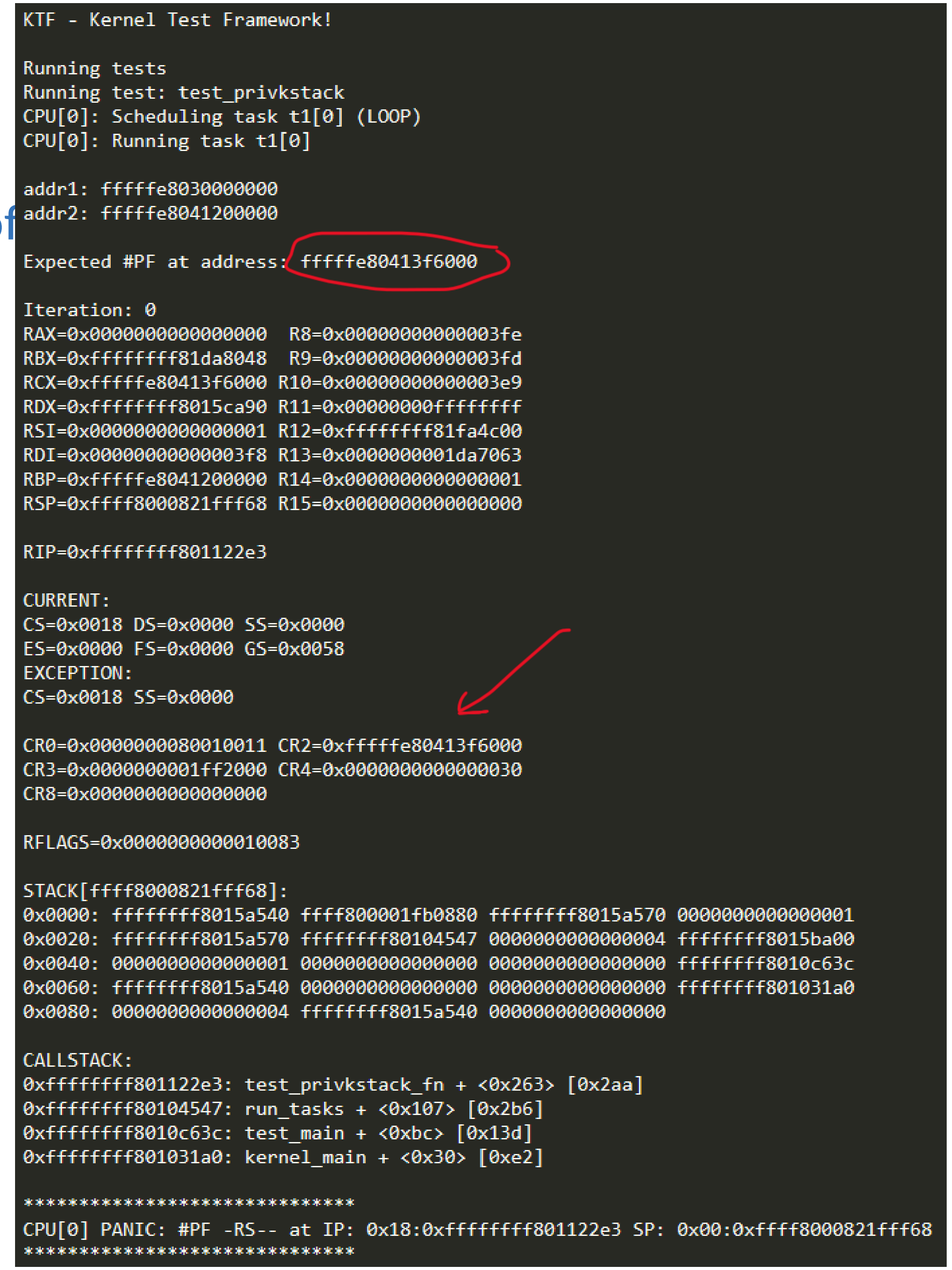
- Finally, by running the main loop, we should be able to trigger the predicted spurious `#PF` exception
    - The loop does the following…
    - At its second iterations, we should hit the paging-structure cache collision…



```c
cli();
while (1) {
    /* Restore original default PDE entry content for 2nd address */
    *pde2 = pde2_bak;
    barrier();

    /* Optional memory and instruction serialization */
    mfence(); sfence(); lfence();

    /* VM-Exit triggering CPUID to flush translation caches in a VM */
    /* cpuid_eax(0x0); */

    /* Accessing entire 2MB virtual address space */
    for (int i = 0; i < NR; i++)
        asm volatile ("mov (%0), %%rax" :: "r" (va2s[i]): "rax");

    printk("Iteration: %u\n", iteration++);

    /* Optional full TLB flush - just in case ;-) */
    flush_tlb();

    /* Temporarily change 2nd level page table of the second virtual
     * address, so it points at the first virtual address space
     */
    *pde2 = *pde1;
    barrier();

    /* Full flush of TLB and Paging-Structure Caches */
    flush_tlb();

    /* Speculatively or not populate TLB with second address translation */
    prefetcht0(addr2);

    /* Optional memory and instruction serialization */
    barrier(); mfence(); sfence(); lfence();
}
```

PaX/grsecurity PRIVATE KSTACKS – Proof

- And there we have it!



```
KTF - Kernel Test Framework!

Running tests
Running test: test_privkstack
CPU[0]: Scheduling task t1[0] (LOOP)
CPU[0]: Running task t1[0]

addr1: fffffe8030000000
addr2: fffffe8041200000

Expected #PF at address: fffffe80413f6000

Iteration: 0
RAX=0x0000000000000000  R8=0x00000000000003fe
RBX=0xffffffff81da8048  R9=0x00000000000003fd
RCX=0xfffffe80413f6000 R10=0x00000000000003e9
RDX=0xffffffff8015ca90 R11=0x00000000ffffffff
RSI=0x0000000000000001 R12=0xffffffff81fa4c00
RDI=0x00000000000003f8 R13=0x000000001da7063
RBP=0xfffffe8041200000 R14=0x0000000000000001
RSP=0xffff8000821fff68 R15=0x0000000000000000

RIP=0xffffffff801122e3

CURRENT:
CS=0x0018 DS=0x0000 SS=0x0000
ES=0x0000 FS=0x0000 GS=0x0058
EXCEPTION:
CS=0x0018 SS=0x0000

CR0=0x0000000080010011 CR2=0xfffffe80413f6000
CR3=0x000000001ff2000 CR4=0x0000000000000030
CR8=0x0000000000000000

RFLAGS=0x0000000000010083

STACK[ffff8000821fff68]:
0x0000: ffffffff8015a540 ffff800001fb0880 ffffffff8015a570 0000000000000001
0x0020: ffffffff8015a570 ffffffff80104547 0000000000000004 ffffffff8015ba00
0x0040: 0000000000000001 0000000000000000 0000000000000000 ffffffff8010c63c
0x0060: ffffffff8015a540 0000000000000000 0000000000000000 ffffffff801031a0
0x0080: 0000000000000004 ffffffff8015a540 0000000000000000

CALLSTACK:
0xffffffff801122e3: test_privkstack_fn + <0x263> [0x2aa]
0xffffffff80104547: run_tasks + <0x107> [0x2b6]
0xffffffff8010c63c: test_main + <0xbc> [0x13d]
0xffffffff801031a0: kernel_main + <0x30> [0xe2]

******************************
CPU[0] PANIC: #PF -RS-- at IP: 0x18:0xffffffff801122e3 SP: 0x00:0xffff8000821fff68
******************************
```

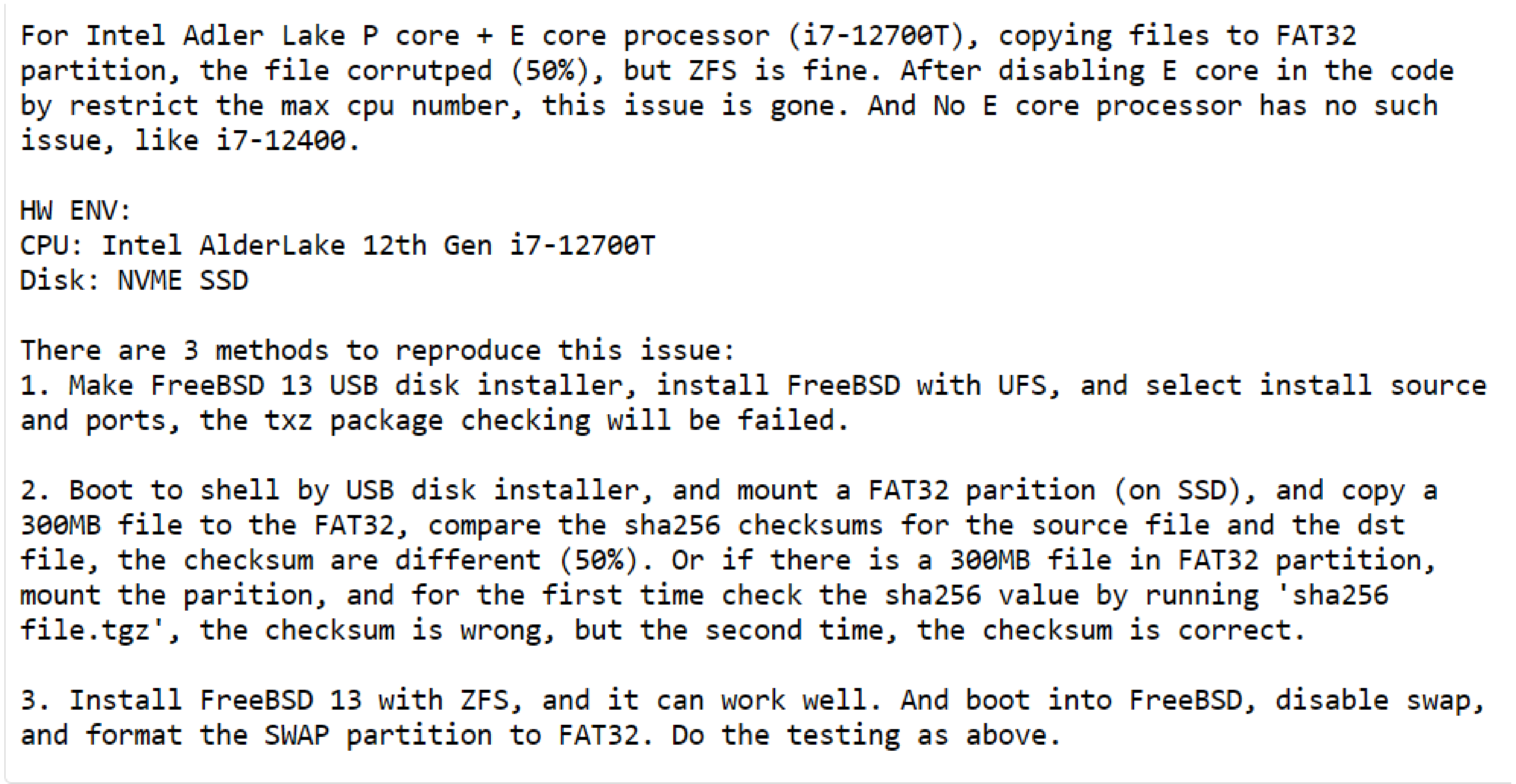
PaX/grsecurity PRIVATE KSTACKS – Proof-of-Concept

- Summary:
  - The higher-level page table entries can be cached in paging-structure caches whenever CPU decides to do so
  - When aliasing and/or collision in translation caches can occur: **invalidation is a must**
    - **That’s why Intel SDM recommends the “at least one INVLPG instruction” invalidation**
  - Otherwise page walk can be optimized and bad things could happen

Broken INVLPG instruction on Intel Gracemont microarchitecture

Broken INVLPG instruction case

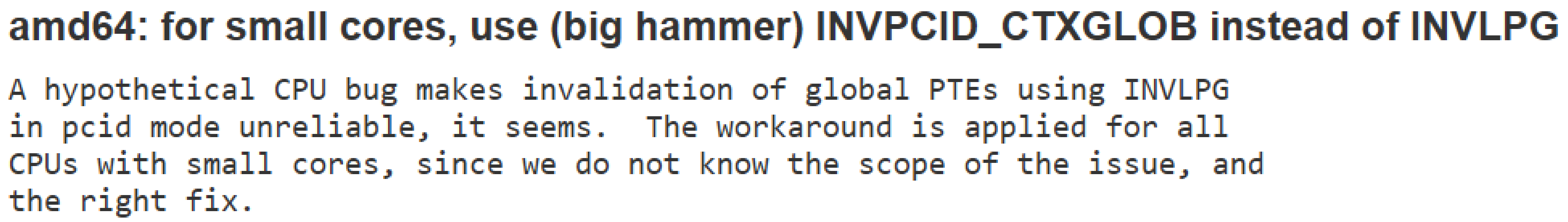
- Around the same time, a FreeBSD received a bug report
  - 261169 – Intel Alder Lake: data corruption with Read&Write files to FAT32 or UFS



```
For Intel Adler Lake P core + E core processor (i7-12700T), copying files to FAT32
partition, the file corrutped (50%), but ZFS is fine. After disabling E core in the code
by restrict the max cpu number, this issue is gone. And No E core processor has no such
issue, like i7-12400.

HW ENV:
CPU: Intel AlderLake 12th Gen i7-12700T
Disk: NVME SSD

There are 3 methods to reproduce this issue:
1. Make FreeBSD 13 USB disk installer, install FreeBSD with UFS, and select install source
and ports, the txz package checking will be failed.

2. Boot to shell by USB disk installer, and mount a FAT32 parition (on SSD), and copy a
300MB file to the FAT32, compare the sha256 checksums for the source file and the dst
file, the checksum are different (50%). Or if there is a 300MB file in FAT32 partition,
mount the parition, and for the first time check the sha256 value by running 'sha256
file.tgz', the checksum is wrong, but the second time, the checksum is correct.

3. Install FreeBSD 13 with ZFS, and it can work well. And boot into FreeBSD, disable swap,
and format the SWAP partition to FAT32. Do the testing as above.
```

Broken INVLPG instruction case

- Some time later FreeBSD received a fix for the issue:
  - https://cgit.freebsd.org/src/commit/?id=cde70e312c3fde5b37a29be1dacb7fde9a45b94a

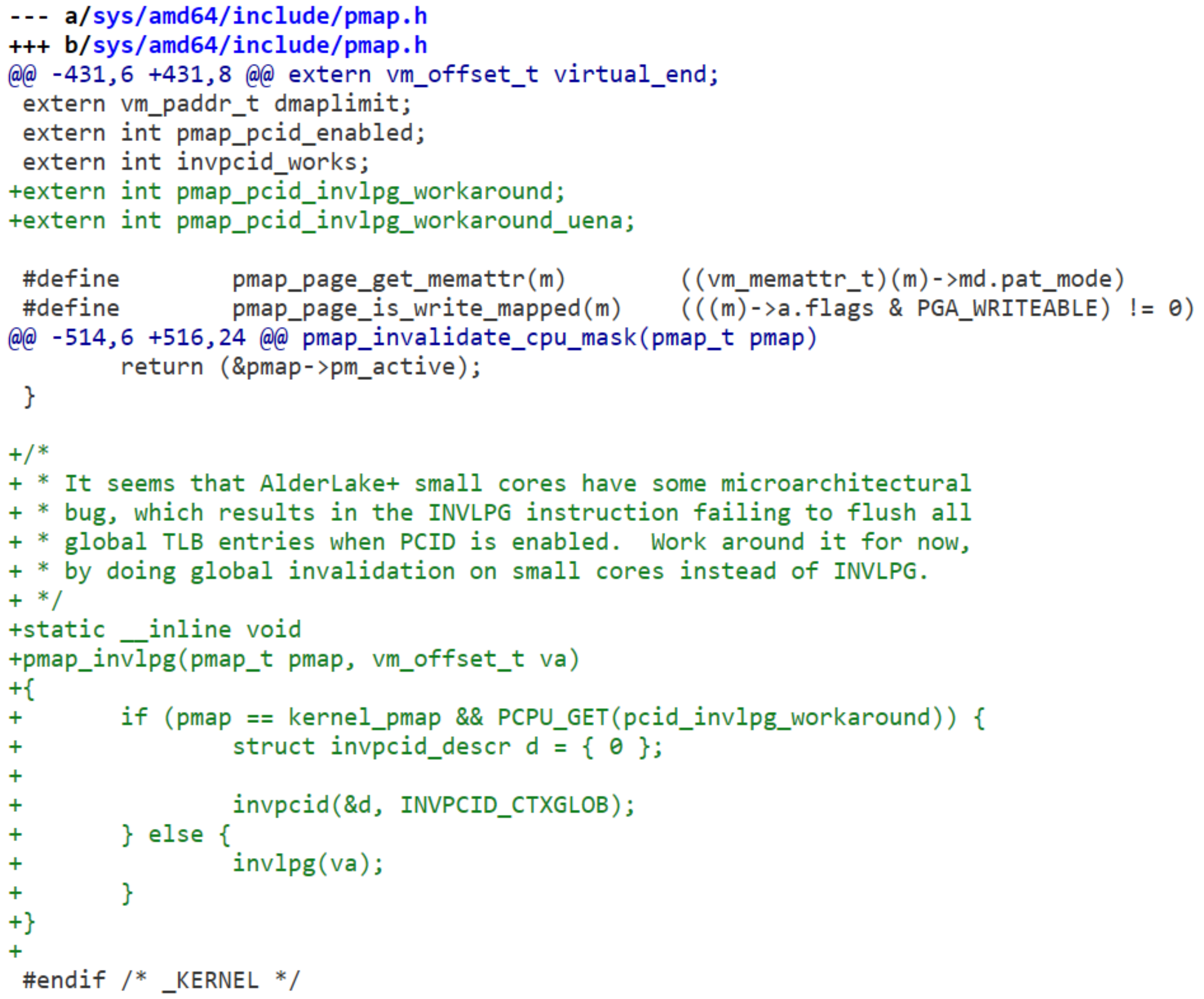


amd64: for small cores, use (big hammer) INVPCID_CTXGLOB instead of INVLPG

```
A hypothetical CPU bug makes invalidation of global PTEs using INVLPG
in pcid mode unreliable, it seems.  The workaround is applied for all
CPUs with small cores, since we do not know the scope of the issue, and
the right fix.
```

Broken INVLPG instruction case

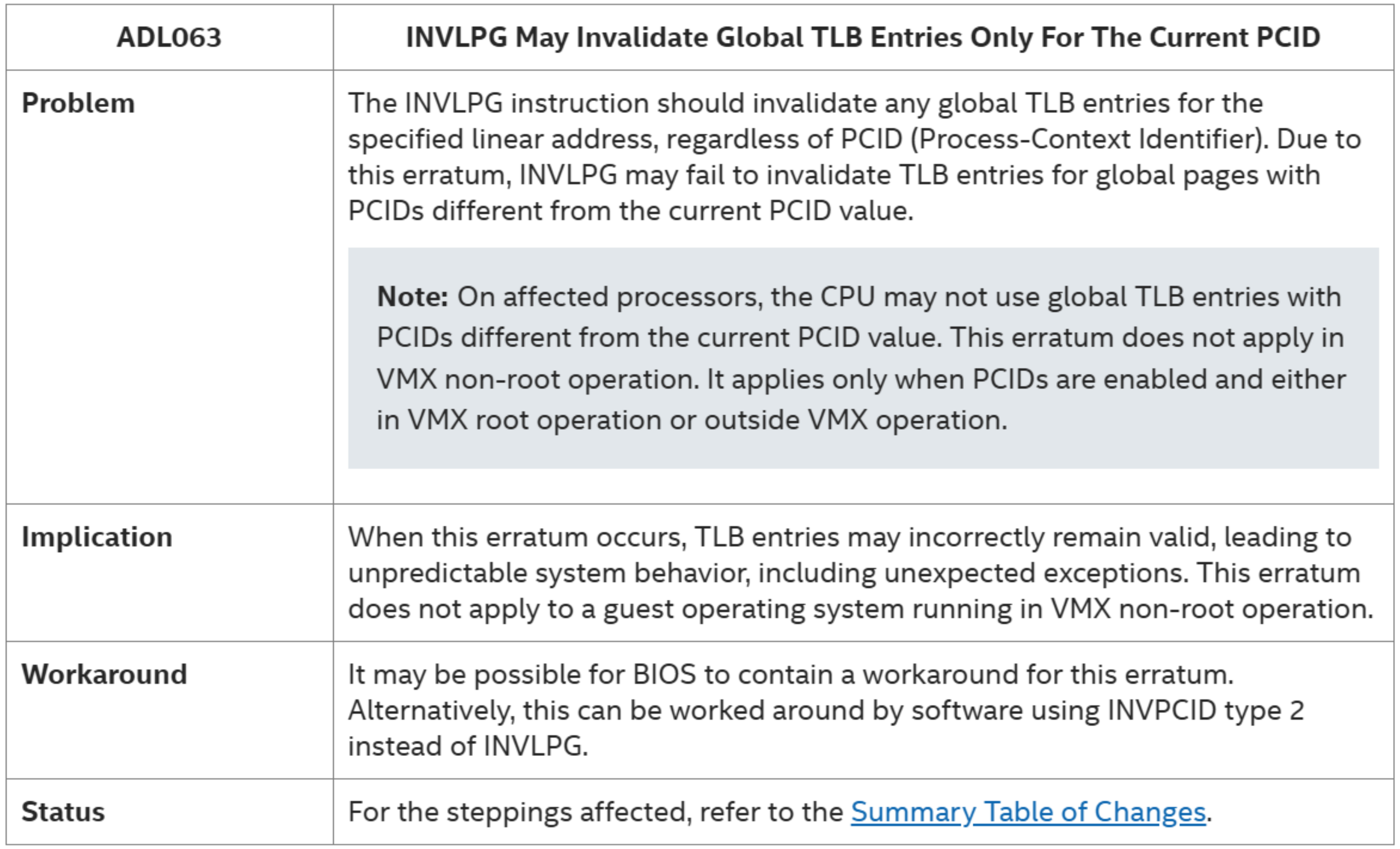
- What does the fix do?
- Replaces `INVLPG` instruction usage with `INVPCID`, because the `INVLPG` instruction apparently does not flush all global TLB entries when PCID is enabled!
    - Despite it should!



```
--- a/sys/amd64/include/pmap.h
+++ b/sys/amd64/include/pmap.h
@@ -431,6 +431,8 @@ extern vm_offset_t virtual_end;
 extern vm_paddr_t dmaplimit;
 extern int pmap_pcid_enabled;
 extern int invpcid_works;
+extern int pmap_pcid_invlpg_workaround;
+extern int pmap_pcid_invlpg_workaround_uena;

 #define        pmap_page_get_memattr(m)       ((vm_memattr_t)(m)->md.pat_mode)
 #define        pmap_page_is_write_mapped(m)   (((m)->a.flags & PGA_WRITEABLE) != 0)
@@ -514,6 +516,24 @@ pmap_invalidate_cpu_mask(pmap_t pmap)
        return (&pmap->pm_active);
 }

+/*
+ * It seems that AlderLake+ small cores have some microarchitectural
+ * bug, which results in the INVLPG instruction failing to flush all
+ * global TLB entries when PCID is enabled.  Work around it for now,
+ * by doing global invalidation on small cores instead of INVLPG.
+ */
+static __inline void
+pmap_invlpg(pmap_t pmap, vm_offset_t va)
+{
+       if (pmap == kernel_pmap && PCPU_GET(pcid_invlpg_workaround)) {
+               struct invpcid_descr d = { 0 };
+
+               invpcid(&d, INVPCID_CTXGLOB);
+       } else {
+               invlpg(va);
+       }
+}
+
 #endif /* _KERNEL */
```

Broken INVLPG instruction case

- Then, Intel publishes an erratum for the affected processors:
  - Errata Details - 027 - ID:682436 | 12th Generation Intel® Core™ Processor



ADL063	INVLPG May Invalidate Global TLB Entries Only For The Current PCID
Problem	The INVLPG instruction should invalidate any global TLB entries for the specified linear address, regardless of PCID (Process-Context Identifier). Due to this erratum, INVLPG may fail to invalidate TLB entries for global pages with PCIDs different from the current PCID value. **Note:** On affected processors, the CPU may not use global TLB entries with PCIDs different from the current PCID value. This erratum does not apply in VMX non-root operation. It applies only when PCIDs are enabled and either in VMX root operation or outside VMX operation.
Implication	When this erratum occurs, TLB entries may incorrectly remain valid, leading to unpredictable system behavior, including unexpected exceptions. This erratum does not apply to a guest operating system running in VMX non-root operation.
Workaround	It may be possible for BIOS to contain a workaround for this erratum. Alternatively, this can be worked around by software using INVPCID type 2 instead of INVLPG.
Status	For the steppings affected, refer to the Summary Table of Changes.

Broken INVLPG instruction case

- The plot thickens…
  - Intel confirmed the incorrect behavior of `INVLPG` instruction, but also …
  - Intel mentioned that the `INVLPG` instruction only fails to flush global TLB entries when PCID is enabled **and** outside of the VMX virtualized environment!

- What was the natural conclusion?
  - That this was the same bug we were having with PRIVATE KSTACKS
    - There were just too many coincidences…

Broken INVLPG instruction case

- But, there was also a couple of serious “buts”…
    - The PCID being enabled or disabled did not play a role for the PRIVATE KSTACKS bug at all
    - And more importantly: unlike FreeBSD, we do not use global pages!

- After a careful investigation we were able to conclude that these were indeed two completely separate issues!
    - One was just a counter-intuitive corner case scenario resulting from specification ambiguity…
        - … and opaque new hardware microarchitectural modifications!
    - The other was just a new hardware CPU bug!

- A classic situation where nothing is what it seems…

Conclusions

- Maintaining coherency between page tables in memory and CPU translation caches is complex
    - Even if only correctness is kept in mind
    - It becomes much harder when optimizing for performance
- Documentation is ambiguous, imprecise and lacks important details
    - Probably on purpose: vendors do not want to reveal implementation details
- Sometimes, something that seems to be a CPU bug, is just a convoluted, counter-intuitive and super rare corner case scenario manifestation
- Other times, the convoluted, counter-intuitive and super rare corner case scenario manifestation is nothing but a CPU bug
- No magic wand

Thank you

wipawel@grsecurity.net

Grsecurity is created by

OPEN
SOURCE
SECURITY