Restricted Kernel Address Spaces

Mike Rapoport
<rppt@linux.ibm.com>

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under grant agreement No 825377

Post Meltdown era

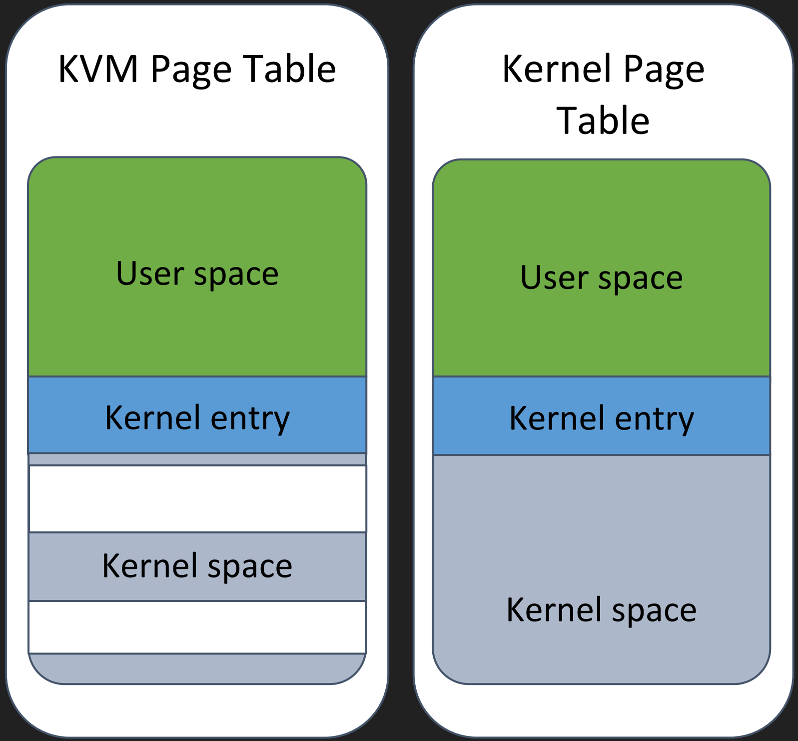
- Speculation vulnerabilities won’t disappear soon
- Address space isolation can be a mitigation
  - PTI, KVM ASI
- Restricting kernel access to memory makes things safer

Restricted mappings in the kernel

- ✓ EFI
- ✓ Page table isolation
- ASI for virtual machines
- Process local memory
- Exclusive user mappings
- KVM protected memory

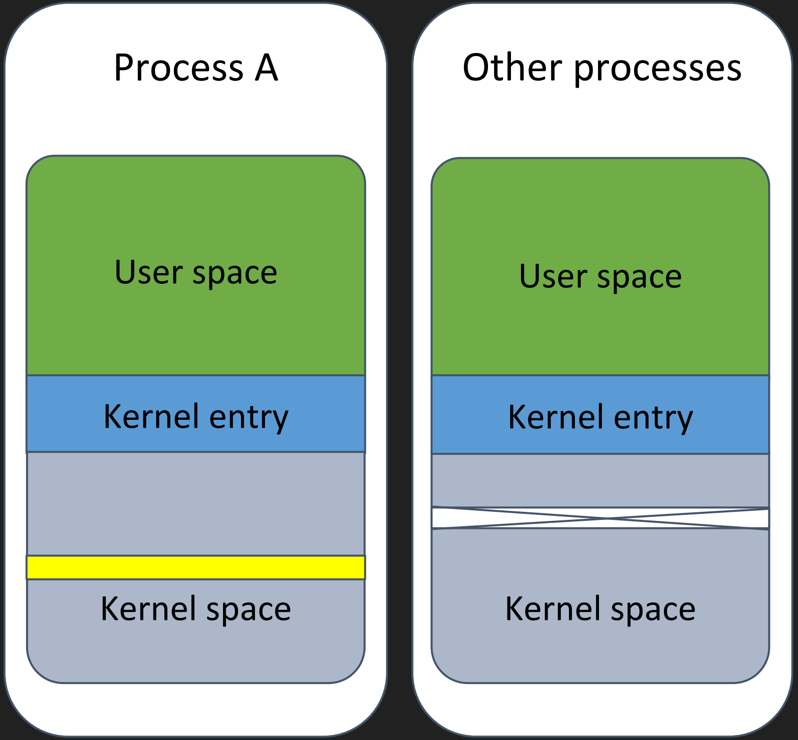
ASI for virtual machines

- Mitigation for L1F and alike with HT enabled
- Restricted context for KVM kernel code



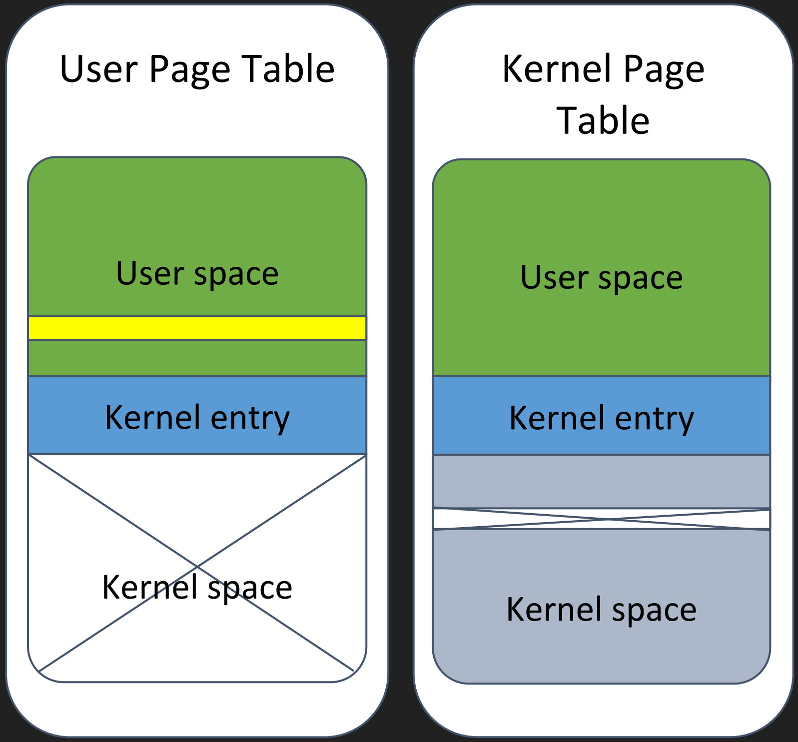
Process local memory

- A variant of `kmalloc()`
- Memory is visible only in the context of a specific process
  - Dropped from the direct map
  - Remapped in a dedicated virtual address range
- Use cases
  - vCPU state, VMCS



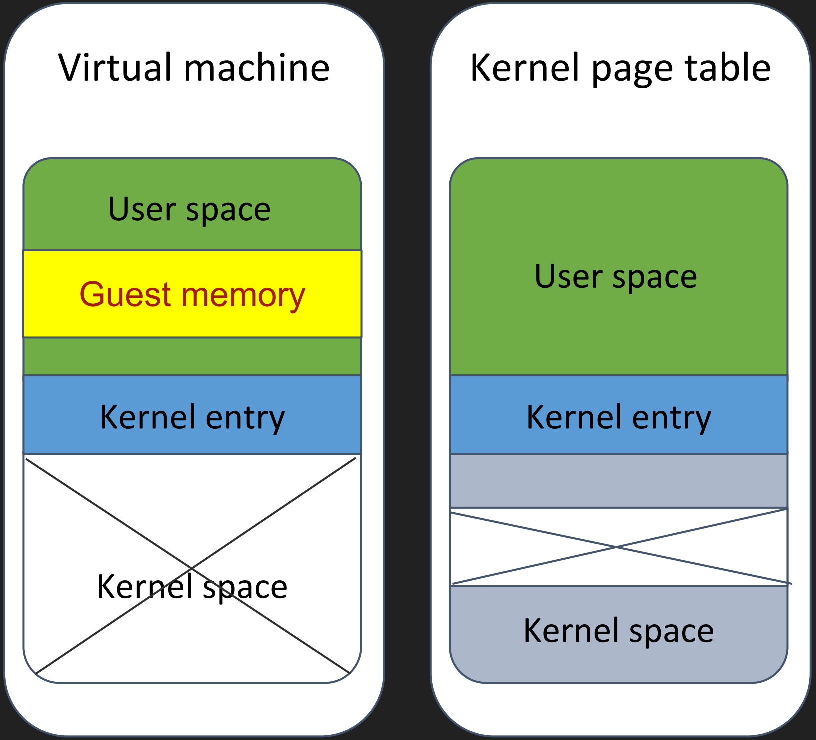
Exclusive user mappings

- Memory region mapped only in a single process page table
  - Excluded from the direct map
- Use-cases
  - Store secrets
  - Protect the entire VM memory



KVM protected memory

- Remove guest memory from the direct map
- Allow hypervisor access in very particular way



Generalizing ASI approach

- **Page table creation and management**
- Context switching
- State tracking

High level API

- Clone page table
  - Similar to copy_page_range()
  - Caller defines what level is shared

```
clone_range(dst, src, va_start, va_end, level)
```

- Map range

```
map_range(dst, virt, phys, prot, nr_pages)
```

- Unmap range

```
unmap_range(dst, virt, nr_pages)
```

Page table representation alternatives

- Use `pXd_t` directly
  - Unfriendly to concurrent updates and tear down
- Use `mm_struct`
  - Most data is dedicated to userspace mm
  - Weird constructs appear

```
clone_range(dst_mm, user_pgd(dst_mm->pgd),
            src_mm, user_pgd(src_mm->pgd))
```

- Add new abstraction for page table

Introduce `struct pg_table`

```
struct pg_table {
        pgd_t           *pgd;
        spinlock_t      page_table_lock;
        atomic_t        pgtables_bytes;
        pt_context_t    pt_context;
        unsigned long cpu_bitmask[];
}
```

```
struct mm_struct {
-       pgd_t pgd;
-       spinlock_t page_table_lock;
-       atomic_t pgtables_bytes;
        ...
+       struct pg_table pgt;
};
```

Introduce `struct pg_table`

- Convert users of `mm->pgd`, `mm->page_table_lock`, ...
  - Use `mm_pgd(mm)`, `mm_pgt(mm)` helpers
  - Can be automated with semantic patch
- Add APIs that operate on `struct pg_table`

```
__foo(struct pg_table *pgt)
{
        /* do stuff */
}

foo(struct mm_struct *mm)
{
        __foo(mm_pgt(mm));
}
```

Introduce `struct pg_table`

- Ensure `PageTable` type is set on all page table pages
  - Important for tear down
  - Allows using two `unsigned long`s in `struct page`
- Easy access to mm_struct for user page tables

```
if (is_user_pgt(pgt)) {
        struct mm_struct *mm =
                        container_of(pgt, mm_struct, pgt);

        bar(mm);
}
```

Introduce `struct pg_table`

- Split fields from mm_context_t to pt_context_t
- Implement context switching for pg_table

```
void switch_pgt(struct pg_table *prev, struct pg_table *next,
                struct task_struct *tsk)
{
        /* do the switch */
}

void switch_mm(struct mm_struct *prev, struct mm_struct *next,
               struct task_struct *tsk)
{
        switch_pgt(&prev->pgt, &next->pgt, tsk);
}
```

Freeing Restricted Page Tables

- Integration with existing TLB management infrastructure
  - Avoid excessive TLB shootdowns
- Special care for shared page table levels
  - Avoid freeing main kernel page tables
- `page::_pt_pad_1` and `page::_pt_pad_2` come handy

Open issues

- Actually set `PageTable` type for page tables
  - Early page tables do not have it
- Placement of `cpu_bitmap`
  - Naturally belongs to `pg_table`, but putting it there taints struct randomization
- Intermix of page table and userspace memory management semantics in `mm_context_t`

Private Memory Allocations

- Extend `alloc_page()` and `kmalloc()` with context awareness
- Pages and objects are visible in a single context
    - Can be a process or all processes in a namespace
- Special care for objects traversing context boundaries

Per-Context Allocations

- Allow per-context allocations
  - `__GFP_EXCLUSIVE` – for pages
  - `SLAB_EXCLUSIVE` – for slabs
- Drop pages from the direct map on allocation, put them back on freeing
  - `set_direct_map_invalid_noflush()`
  - `set_direct_map_default_noflush()`
- Need for synchronization of all page tables

Marking pages in restricted mappings

- New type for kernel pages
  - `PageFromRestrictedContext`
- Hide user pages behind anonymous inode
  - Similar to anonymous HugeTLB
  - Differentiate using `page->mapping`

Direct map fragmentation

- Direct map uses 2nd and 3rd level leaf pages
  - 1G and 2M on x86
- Removing pages from the direct map fragments it
  - s/1G page/512 2M pages/ s/2M page/512 4K pages/
  - Performance degradation

Keeping large pages in the direct map

- Preallocate memory at boot and manage it separately
  - Similar to mem=X
  - Kernel still can access memory with `gup()`/`kmap()` like APIs
- Use local pools of large pages
  - Exclusive user mappings, SL*Bs
- Add direct map layout awareness to page allocator

Large pages in the direct map

- Support for 4M pages for Pentium CPU
  - Version 1.3.16 (1995)

```
+ pgd_val(pg_dir[0]) = _PAGE_TABLE | _PAGE_4M | address;
```

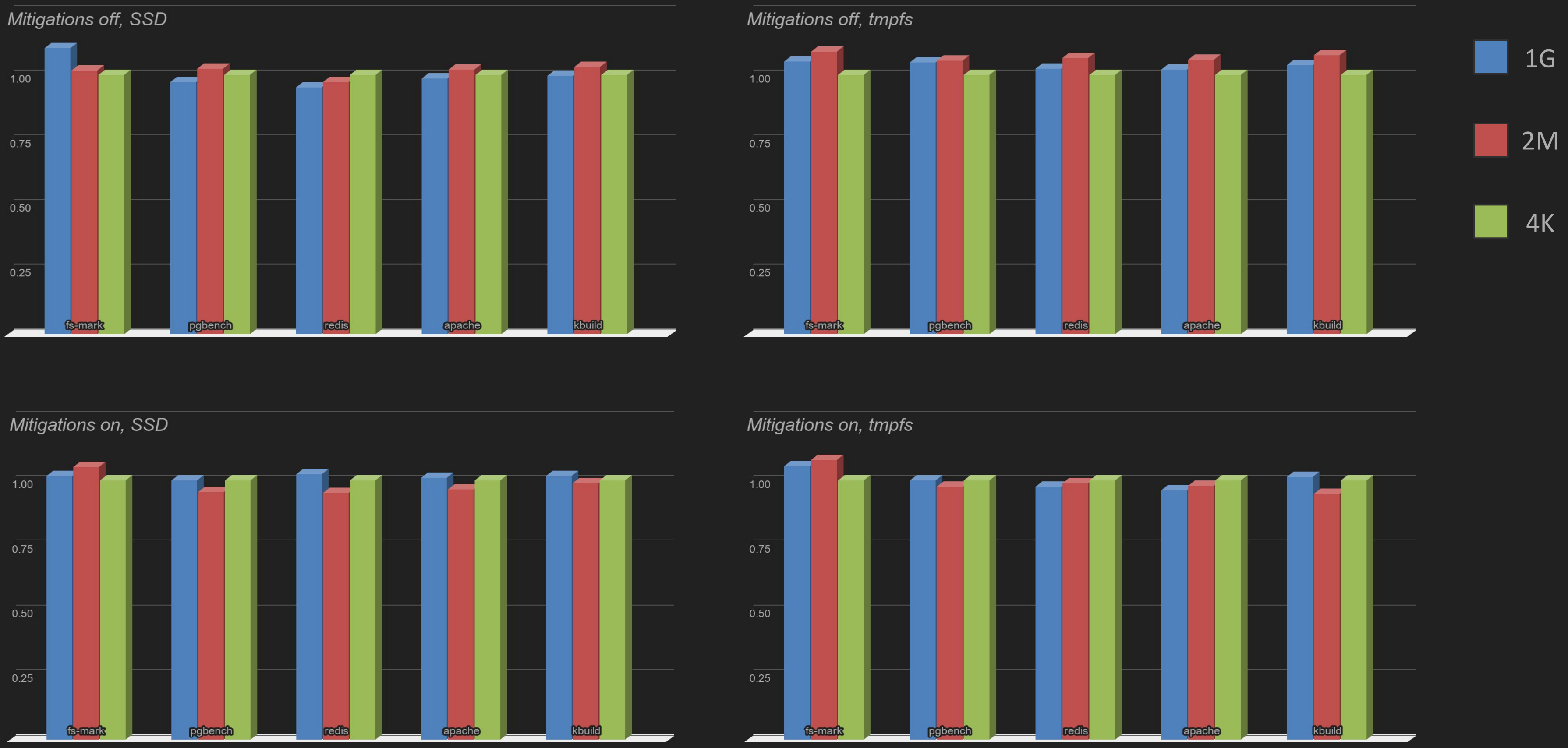
- Support for 1G pages for AMD Fam10h CPU

```
commit ef9257668e3199f9566dc4a31f5292838bd99b49
Author: Andi Kleen <ak@suse.de>
Date:   Thu Apr 17 17:40:45 2008 +0200

        x86: do kernel direct mapping at boot using GB pages

The AMD Fam10h CPUs support new Gigabyte page table entry for
mapping 1GB at a time. Use this for the kernel direct mapping.
```

Direct map fragmentation

- ThinkPad T480
  - i7-8650U CPU @ 1.90GHz
  - 32G RAM, WDC SN720 SSD
- Benchmarks
  - FS-mark, pgbench, redis, apache, kbuild
- Configurations
  - Force the entire direct map to 4K or 2M pages
  - SSD vs tmpfs
  - mitigations=off vs mitigations=on

Direct map fragmentation



Conclusion

- Using restricted contexts improves security
- Reworking kernel address space management is a major challenge
- Direct map fragmentation is not a disaster

References

- ASI RFC v4
  https://lore.kernel.org/lkml/20200504144939.11318-1-alexandre.chartre@oracle.com/
  https://lore.kernel.org/lkml/20200504145810.11882-1-alexandre.chartre@oracle.com/
  https://lore.kernel.org/lkml/20200504150235.12171-1-alexandre.chartre@oracle.com/
- Proclocal
  https://lore.kernel.org/lkml/20190612170834.14855-1-mhillenb@amazon.de/
- Exclusive user mappings
  https://lore.kernel.org/lkml/20200818141554.13945-1-rppt@kernel.org/
- KVM Protected memory
  https://lore.kernel.org/lkml/20200522125214.31348-1-kirill.shutemov@linux.intel.com

References

- `struct pg_table`
  https://git.kernel.org/pub/scm/linux/kernel/git/rppt/linux.git/log/?h=pg_table/v0.0
- 4M pages for Pentuim CPU
  https://github.com/mpe/linux-fullhistory/commit/10a137bfab8acd637fe98a74c5d3d7b331b67dc8#diff-f3ec8be0f5e88a2c3dff2d2b2b4fdb93
- 1G pages for AMD Fam10h CPU
  https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ef9257668e3199f9566dc4a31f5292838bd99b49
- Benchmarks
  https://docs.google.com/spreadsheets/d/1tdD-cu8e93vnfGsTFxZ5YdaEfs2E1GELlvWNOGkJV2U/edit?usp=sharing

Thank you!