

# KVM ASI

## (Address Space Isolation)

Liran Alon  
Alexandre Chartre


KVM Forum 2019

# Oracle - Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, timing and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

# Who are we?

- **Liran Alon**

- \* Architect at OCI (Oracle Cloud Infrastructure)
- \* 5 years of Virtualization, SDN and Cloud Computing
- \* 7 years of cyber security RnD
- \* Active KVM contributor: x86, VMX, SVM, nested-virtualization, QEMU, SeaBIOS
- \* Interests: Anything Low-Level
- \*  @Liran\_Alon

- **Alexandre Chartre**

- \* Senior Principal Engineer at Oracle Linux & Virtualization group
- \* Over 20 years experience in kernel development and virtualization

# Agenda

- Background
  - L1TF Hyperthreading attack scenario
- Community mitigation mechanisms
- KVM ASI
- Future mitigations

Background

# Fault Speculative Execution

```
value = *ptr; ← Read from inaccessible memory.  
x = value + 1; Should raise #PF.  
y = x * 2;  
...
```

# Fault Speculative Execution

value = \*ptr;

x = value + 1;

y = x \* 2;

...

← Until faulting instruction retire, CPU **speculatively** execute following instructions out-of-order.

**Speculate** \*ptr value based on **micro-architecture cache**. Continue execution based on speculated value.

# Fault Speculative Execution

value = \*ptr;

x = value + 1;

y = x \* 2;

...

← Until faulting instruction retire, CPU **speculatively** execute following instructions out-of-order.

**Speculate** \*ptr value based on **micro-architecture cache**. Continue execution based on speculated value.



# Fault Speculative Execution

```
value = *ptr;
```

```
x = value + 1;
```

```
y = x * 2;
```

```
...
```



**Read from \*ptr retired.**

- 1) Abort speculation (Drop changes to x & y)**
- 2) Raise #PF on faulting instruction**

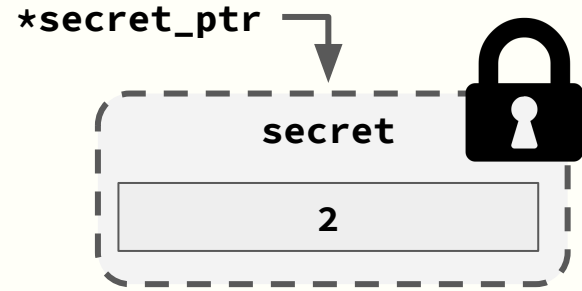
# Fault Speculative Execution Vulnerability

- Faults are raised only when faulting instruction retire
- Until then, the CPU **speculatively** execute following instructions out-of-order
- **Which can compute on unauthorized results of the faulting instruction**
- Should be OK as speculation **aborts** when faulting instruction retire

# Fault Speculative Execution Vulnerability

- Faults are raised only when faulting instruction retire
- Until then, the CPU **speculatively** execute following instructions out-of-order
- **Which can compute on unauthorized results of the faulting instruction**
- Should be OK as speculation **aborts** when faulting instruction retire
- However, speculation abort **don't revert various micro-arch effects**...
- Which can be measured to deduce unauthorized result

# Fault Speculative Execution Vulnerability Exploit



`probe_array[]`

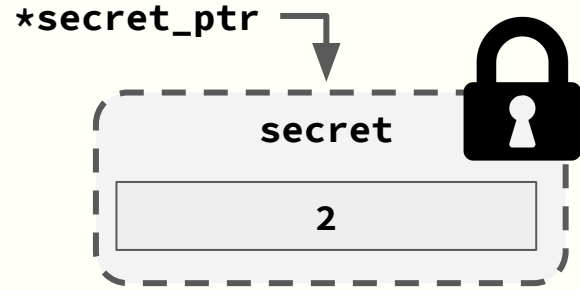
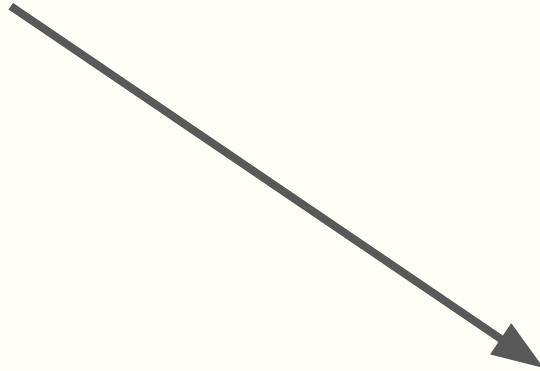


....



# Fault Speculative Execution Vulnerability Exploit

1. Clear `probe_array[]` from cache



`probe_array[]`

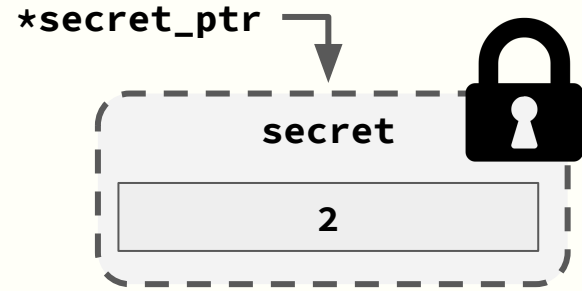


....

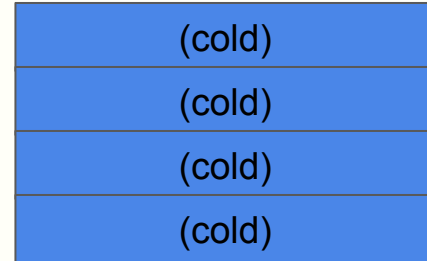


# Fault Speculative Execution Vulnerability Exploit

```
2. i = *secret_ptr;  
   dummy = probe_array[i];
```



`probe_array[]`



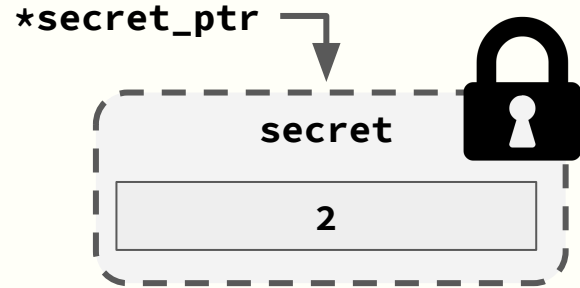
....



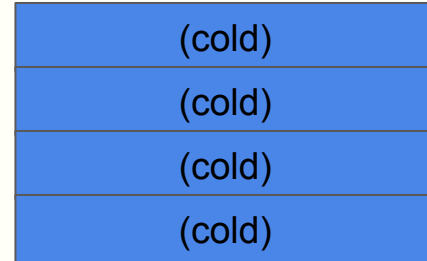
# Fault Speculative Execution Vulnerability Exploit

```
2. i = *secret_ptr;  
   dummy = probe_array[i];
```

- Read from **inaccessible** memory
- Should raise #PF
- #PF will be raised when instruction retire
- Until then, **speculatively** continue execution



`probe_array[]`

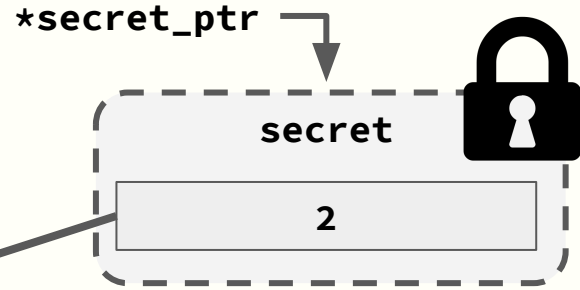


...



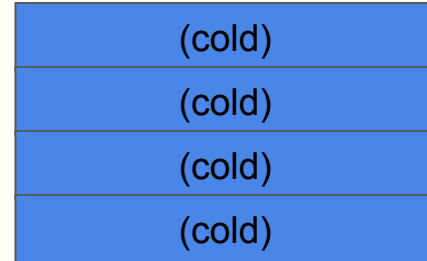
# Fault Speculative Execution Vulnerability Exploit

```
2. i = *secret_ptr;  
   dummy = probe_array[i];
```



**2**

`probe_array[]`



...

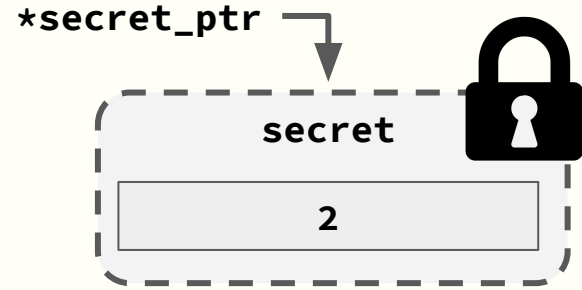


- Read secret speculatively
- Based on some **micro-arch cache**



# Fault Speculative Execution Vulnerability Exploit

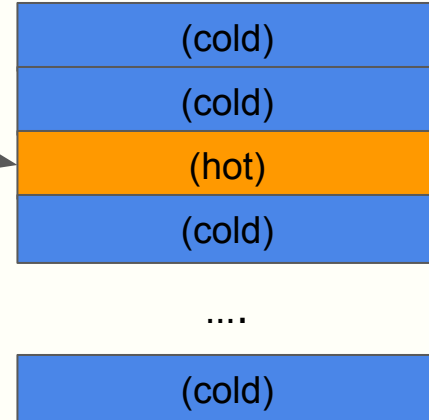
```
2. i = *secret_ptr;  
   dummy = probe_array[i];
```



**2**

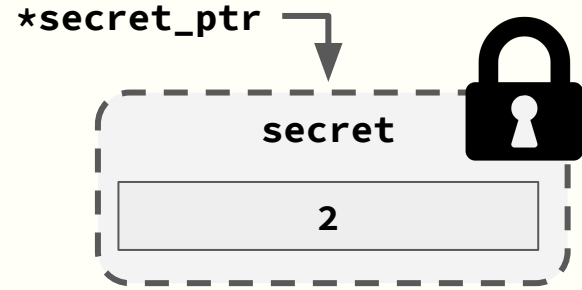
- Read `probe_array[2]` speculatively
- Fetch `probe_array[2]` to cache

`probe_array[]`



# Fault Speculative Execution Vulnerability Exploit

3. When faulting instruction retire:
  - a) **Abort** speculation (Drop changes)
  - b) Raise #PF on faulting instruction



`probe_array[]`



...



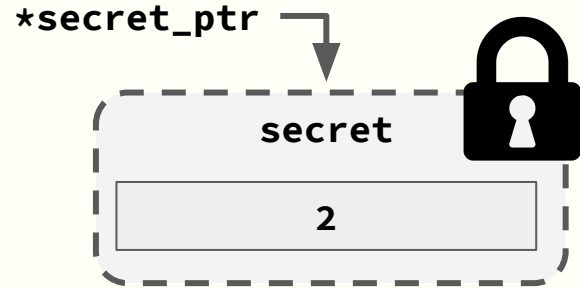
# Fault Speculative Execution Vulnerability Exploit

- When faulting instruction retire:
  - Abort** speculation (Drop changes)
  - Raise #PF on faulting instruction

## Note:

**probe\_array[2] remains hot in cache!**

**Abort doesn't revert micro-arch cache state**



`probe_array[]`

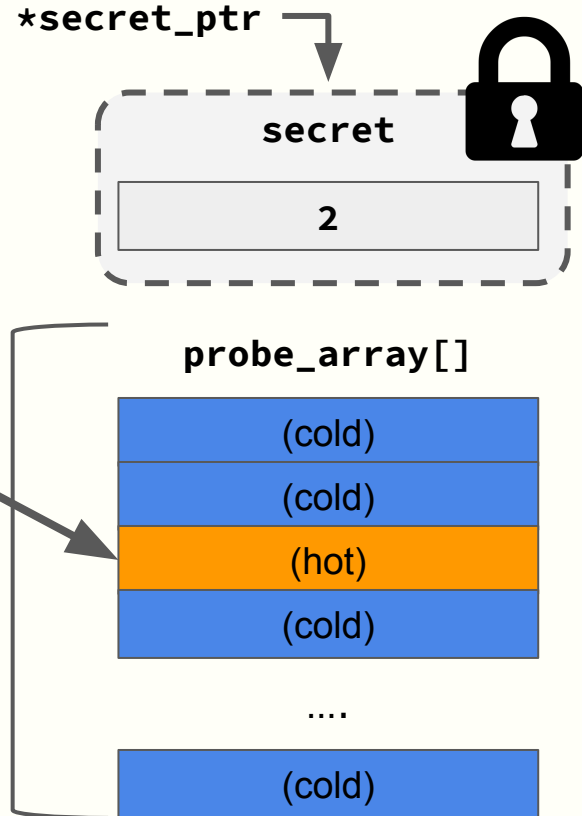


...



# Fault Speculative Execution Vulnerability Exploit

4. Catch #PF
5. Measure access time of probe\_array[]  
⇒ Fastest element is hot in cache  
⇒ **Deduce that secret==2**



# L1TF (L1 Termination Fault)

```
secret = *ptr; ← What if ptr mapped non-present in PTE?  
dummy = probe_array[secret];
```

\*ptr PTE:

PFN	Flags (P=0)
-----	-------------

**Should raise #PF...**

# L1TF (L1 Termination Fault)

```
secret = *ptr;  What if ptr mapped non-present in PTE?  
dummy = probe_array[secret];
```

\*ptr PTE:

PFN	Flags (P=0)
-----	-------------

**But until then...**

**Speculate** value based on **L1D\$[PFN]**



# L1TF (L1 Termination Fault)

```
secret = *ptr;  What if ptr mapped non-present in PTE?  
dummy = probe_array[secret];
```

\*ptr PTE:

PFN	Flags (P=0)
-----	-------------

**But until then...**

**Speculate** value based on **L1D\$[PFN]** 🤖

**Even worse: Doesn't translate PFN via EPT!** 🤖🤖

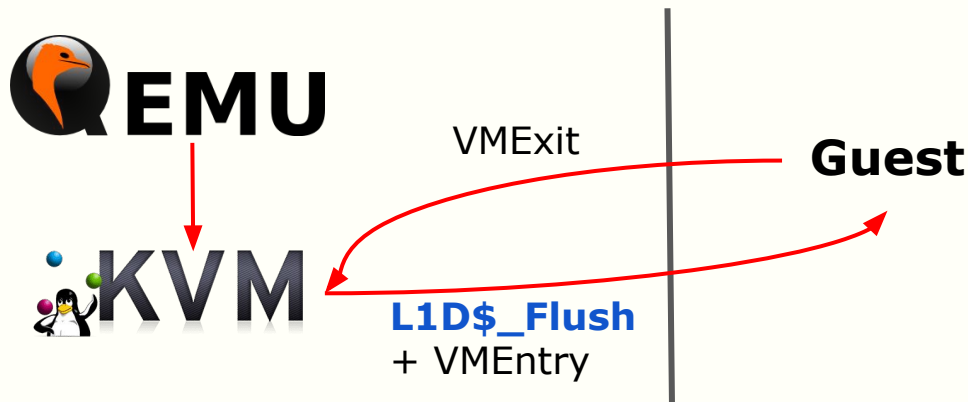
# L1TF: Guest→Host Attack

- Guest kernel completely controls guest page-tables PTEs
- ⇒ Guest can easily trigger L1TF to leak any host data present in L1D cache



# L1TF: Guest→Host Attack

- Guest kernel completely controls guest page-tables PTEs
- ⇒ Guest can easily trigger L1TF to leak any host data present in L1D cache
  
- Mitigation: CPU microcode mechanism to explicitly flush L1D cache
- VMM modified to flush L1D cache before every entry to guest

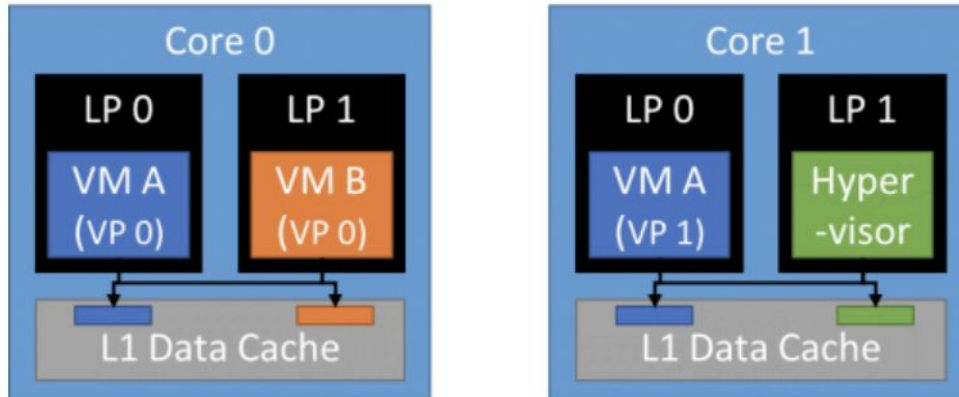


# L1TF: Guest→Host Attack

- Guest kernel completely controls guest page-tables PTEs
- ⇒ Guest can easily trigger L1TF to leak any host data present in L1D cache
  
- Mitigation: CPU microcode mechanism to explicitly flush L1D cache
- VMM modified to flush L1D cache before every entry to guest
  
- Alternative mitigation: Use shadow MMU
  - Allows VMM to completely control PFNs used in non-present PTEs when guest is running
  - VMM will set invalid PFN in non-present PTEs (e.g. MSB bit set)

# L1TF: Hyperthreading

- But flushing L1D cache (L1D\$) on VMEntry is insufficient!
- **L1D\$ is shared** between hyperthreads on same CPU core
- **vCPU in guest can leak L1D\$ data populated by sibling hyperthread**
  - \* Sibling hyperthread runs vCPU thread of another VM
  - \* Sibling hyperthread runs vCPU thread currently running in host (#VMExit)



Source:  
<https://techcommunity.microsoft.com/t5/Virtualization/Hyper-V-HyperClear-Mitigation-for-L1-Terminal-Fault/ba-p/382429>

# L1TF: Disable Hyperthreading?

- Disabling hyperthreading is a valid mitigation...
- But it doesn't fit well all production use-cases
- For example, a public cloud will lose half of it's Compute fleet capacity
  - Leads to losing \$\$\$
  - Renders this mitigation problematic for cloud usage

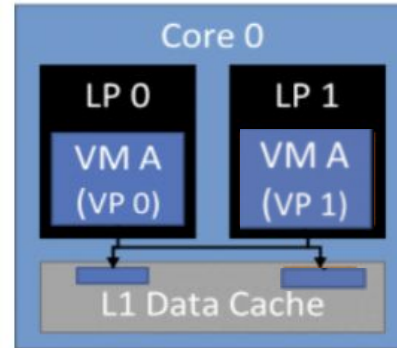
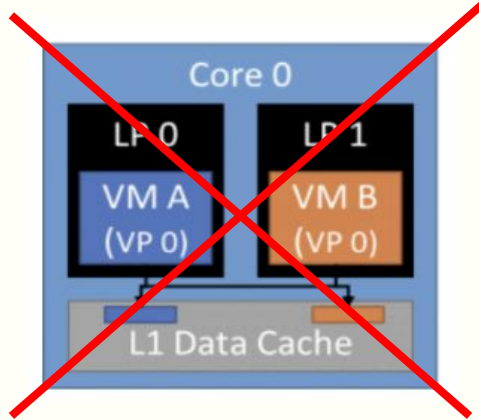
# Community Mitigation Mechanisms

# Community Mitigation Mechanisms

- Multiple mitigation mechanisms were suggested upstream
- None have yet been integrated into upstream Linux

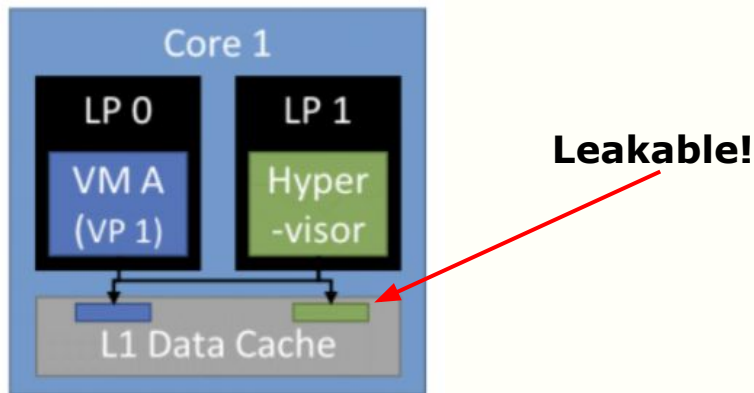
# Core-Scheduling

- A new scheduler policy
- Allows tagging tasks which can run as sibling hyperthreads
- Can be used to guarantee sibling hyperthreads run vCPUs of same VM
- Patches submitted upstream (Peter Zijlstra)



# Remove sensitive data from KVM VA space

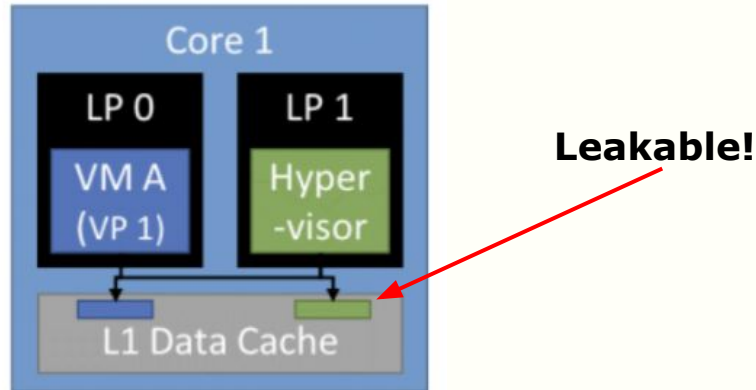
- Hyperthread in guest can still leak from sibling running #VMExit handler





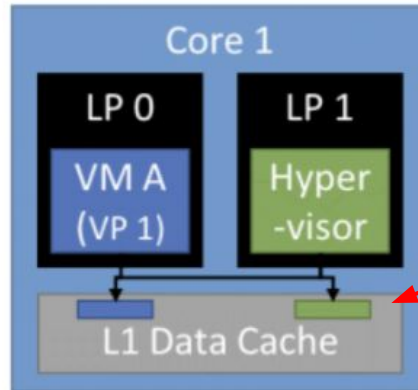
# Remove sensitive data from KVM VA space

- Hyperthread in guest can still leak from sibling running #VMExit handler
- Any executed cache-load gadget sufficient to fill L1D\$
  - E.g. #VMExit handler executes “val = host\_array[guest\_controlled\_index];”
  - **Even if executed speculatively (May allow to read out-of-bounds)**



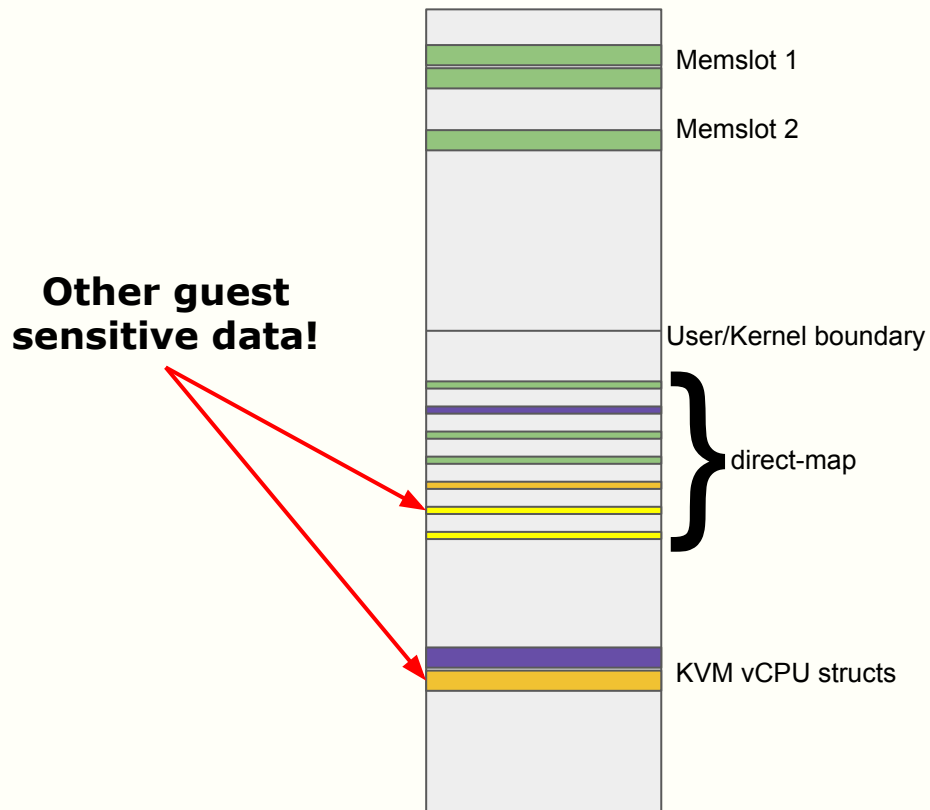
# Remove sensitive data from KVM VA space

- Hyperthread in guest can still leak from sibling running #VMExit handler
- Any executed cache-load gadget sufficient to fill L1D\$
  - E.g. #VMExit handler executes “val = host\_array[guest\_controlled\_index];”
  - Even if executed speculatively (May allow to read out-of-bounds)
- ⇒ **Introduce mechanisms to remove sensitive data from KVM VA space**
  - Assume data not mapped in VA space cannot be fetched into L1D\$
- KVM is a Type-2 hypervisor, thus this includes all host kernel VA space



**Leakable!**  
**But not sensitive!**

# KVM vCPU task VA Space



# XPFO (eXclusive Page Frame Ownership)

- Idea: Remove physical pages used only by userspace from “direct-map”
  - Patches submitted upstream (Juerg Haefliger)
- Originally aimed to mitigate SMAP bypass using “direct-map”
- As a side-effect, **removes guest memory from “direct-map” VA region**

# XPFO (eXclusive Page Frame Ownership)

- Idea: Remove physical pages used only by userspace from “direct-map”
  - Patches submitted upstream (Juerg Haefliger)
- Originally aimed to mitigate SMAP bypass using “direct-map”
- As a side-effect, **removes guest memory from “direct-map” VA region**
- **Performance hit currently too high: ~5%**
  - Cannot map “direct-map” with 1GB PTEs
  - Frequent allocation/free of user-mode pages result in frequent TLB invalidations

# Process local kernel VA region

- Idea: Portion of kernel VA space will be mapped differently between tasks
  - Usually all tasks maps kernel VA space exactly the same
  - Patches submitted upstream (Marius Hillenbrand & Julian Stecklina from AWS)
- Declare a single PGD entry that maps differently between tasks

# Process local kernel VA region

- Idea: Portion of kernel VA space will be mapped differently between tasks
  - Usually all tasks maps kernel VA space exactly the same
  - Patches submitted upstream (Marius Hillenbrand & Julian Stecklina from AWS)
- Declare a single PGD entry that maps differently between tasks
- KVM will **use this VA region to put per-VM sensitive data**
- ⇒ **Unmapping this data from KVM VA space of vCPU threads of other VMs**
  - KVM per-VM & per-vCPU structures
  - Temp map of guest RAM during emulation (e.g. VMPTRLD, MMIO)
  - kmap of guest RAM by vhost kernel backend
  - **Note: Physical pages holding data could still be mapped in direct-map...**

# KVM ASI (Address Space Isolation)



# KVM ASI (Address Space Isolation)

- Previous mitigations remove data from KVM VA space by **blacklist**
  - i.e. Explicitly filtered-out everything we spotted as “sensitive”
  - Difficult to identify all sensitive data
- A more secure approach is to build KVM VA space by **whitelist**
  - Explicitly select non-sensitive & per-vCPU data to be mapped in KVM VA space
  - **Decouple KVM VA space from host VA space**

# KVM ASI (Address Space Isolation)

- Previous mitigations remove data from KVM VA space by **blacklist**
  - i.e. Explicitly filtered-out everything we spotted as “sensitive”
  - Difficult to identify all sensitive data
- A more secure approach is to build KVM VA space by **whitelist**
  - Explicitly select non-sensitive & per-vCPU data to be mapped in KVM VA space
  - **Decouple KVM VA space from host VA space**
- ⇒ **KVM ASI builds a separate VA space used by KVM #VMExit handlers**
- Initially introduced in KVM Forum 2018 BoF session by Liran Alon
  - Inspired by Microsoft Hyper-V HyperClear L1TF mitigation
- Implementation mostly done by Alexandre Chartre

# KVM ASI

- Create VA space that maps only per-VM info, KVM and core kernel mappings
  - Core kernel mappings == kernel text, IDT/GDT, current stack, enter/exit IRQ and etc.
- Use VA space when entering guest
- **Majority** of #VMExits are fully handled in this isolated VA space

# KVM ASI

- Create VA space that maps only per-VM info, KVM and core kernel mappings
  - Core kernel mappings == kernel text, IDT/GDT, current stack, enter/exit IRQ and etc.
- Use VA space when entering guest
- **Majority** of #VMExits are fully handled in this isolated VA space
- When #VMExit requires data outside of isolated VA space:  
**Kick sibling hyperthreads**, switch to full VA space & sync enter to guest
  - Can be done by switching task core-scheduler security-domain (“core-cookie”)
  - Need to exit isolated VA space to handle async events (e.g. interrupts)

# KVM ASI

- Create VA space that maps only per-VM info, KVM and core kernel mappings
  - Core kernel mappings == kernel text, IDT/GDT, current stack, enter/exit IRQ and etc.
- Use VA space when entering guest
- **Majority** of #VMExits are fully handled in this isolated VA space
- When #VMExit requires data outside of isolated VA space:  
**Kick sibling hyperthreads**, switch to full VA space & sync enter to guest
  - Can be done by switching task core-scheduler security-domain (“core-cookie”)
  - Need to exit isolated VA space to handle async events (e.g. interrupts)
- **No need to flush L1D\$** on VMEntry if haven't left isolated VA space

# KVM Isolated VA Space Content

- Core kernel mappings
  - Kernel text, GDT, current stack
  - Minimum mappings to enter/exit ASI
- Additional kernel mappings
  - Stack canary, CPU offsets, current task, rcu data, hw\_events
- KVM specific mappings
  - KVM modules, srcu, current\_vcpu
  - per-vcpu (or per-VM) data: kvm\_vmx, vmx, vcpu, vmcs
  - KVM memslots, buses

# Kernel ASI

- KPTI uses isolated VA space when running user-mode code
- KPTI & KVM ASI is very similar:
  - \* Explicit enter to isolated VA space
  - \* Explicit exit from isolated VA space
  - \* Implicit exit isolated VA space on async event (e.g. Interrupt)
- ⇒ **Should we consider a unified framework for creating Kernel ASI?**

# Kernel ASI: Implementation

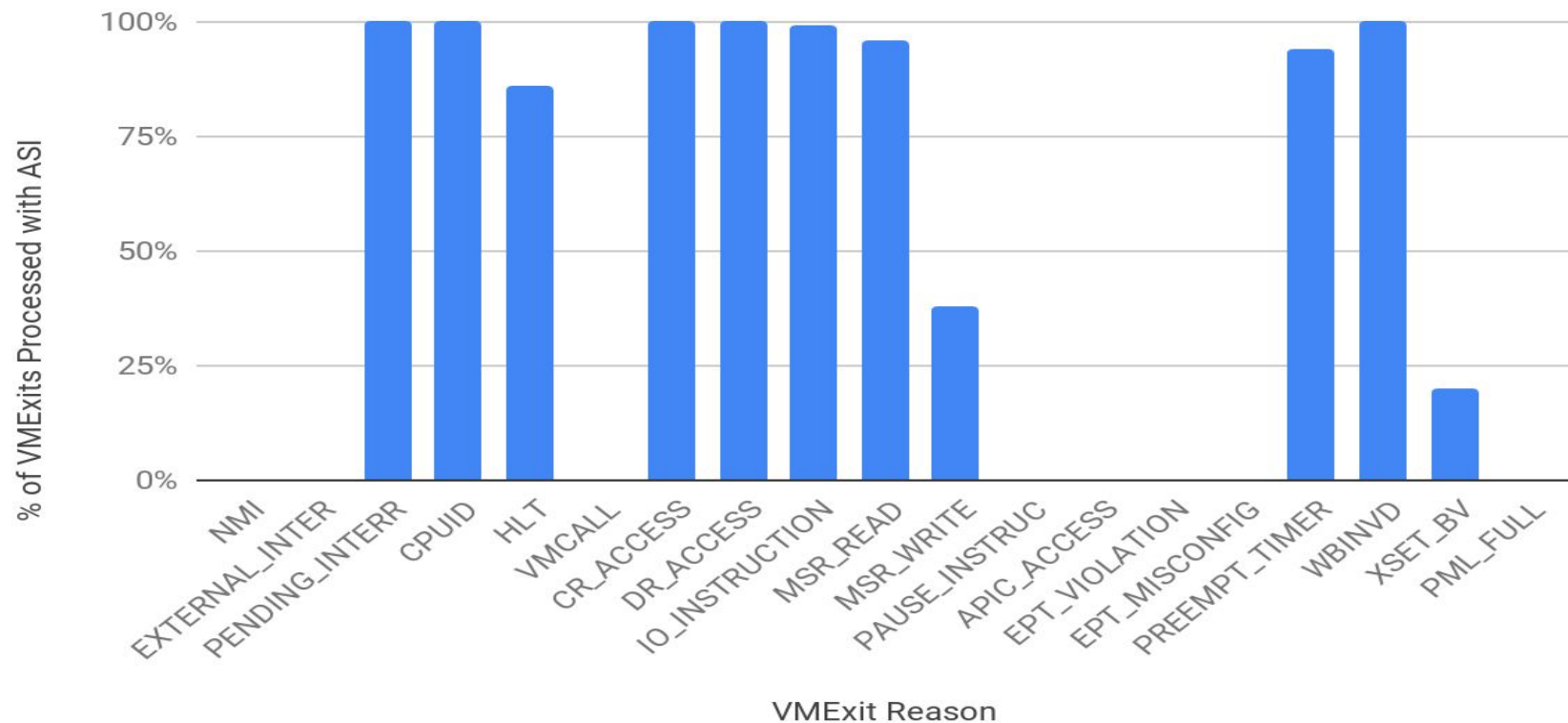
- Mechanism to switch page table
- Mechanism for explicit enter/exit ASI transition
  - E.g. Syscall return for KPTI, VMEntry for KVM
- Mechanism for implicit enter/exit ASI transition to handle async events
  - Interrupt, Exception, NMI, MCE and context-switch
  - Includes mechanism to create page table with min mappings to handle these async events
- Extend `alloc_page()` & `kmalloc()` with context awareness
  - Drop per-context pages from “direct-map”



# KVM ASI: Preliminary Performance Data

- Very preliminary and limited performance data
  - Data collected during VM boot
  - No data with VM running workload (system currently hangs under load)
- `vcpu_run()` loop
  - ASI is entered in `vcpu_enter_guest()` (when needed)
  - >97% of loop iterations are done without exiting ASI
  - Only 75% for nested VM
- Most common `#VMExits` are almost always processed without exiting ASI
  - Kernel VMExit processing (`kvm_vmx_exit_handlers[]()` calls)
  - >99% of `CR_ACCESS`, `CPUID`, `IO_INSTRUCTION`
  - Represent >98% of VMExits
- Probably can be further improved by mapping additional data to ASI

# Kernel VMExit ASI Processing



# Kernel ASI: Current Status

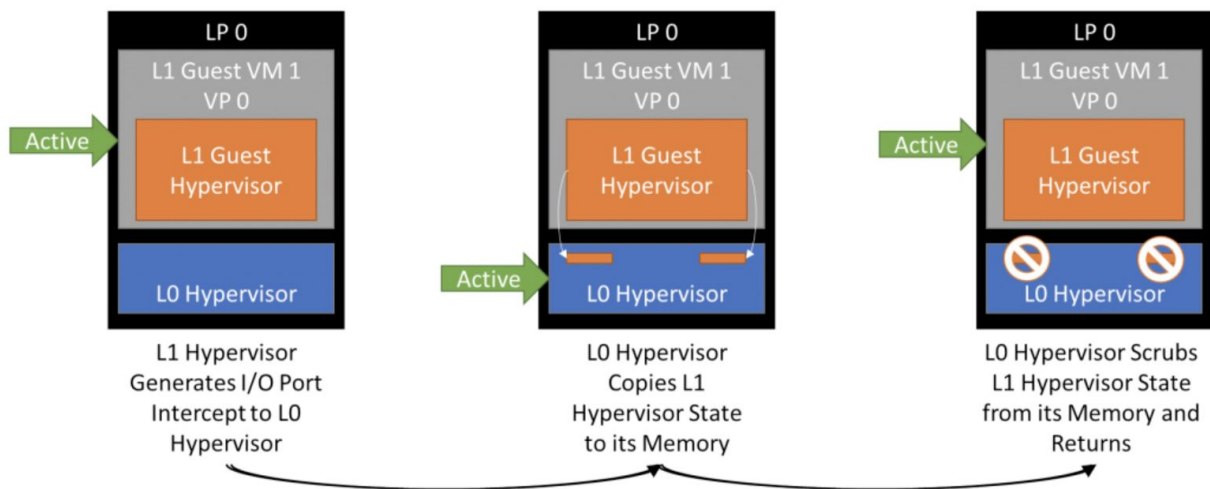
- Patch series v1 & v2 are submitted upstream
  - v1 (KVM ASI): <https://lkml.org/lkml/2019/5/13/515>
  - v2 (Kernel ASI): <https://patchwork.kernel.org/project/kvm/list/?series=144811>
- Doesn't yet kick sibling on ASI exit
  - Need to re-evaluate performance degradation after integration
- Still have unresolved issues
  - Stability (hang under load)
  - Identify all that is required to be mapped in isolated VA space
  - Prefer to just decouple KVM VA space from host and iterate on removed mappings?
    - Not whitelist. However, still decoupled which allows modifications without affecting host.
    - E.g. Remove direct-map from KVM VA space

# ASI & Nested-Virtualization

- ASI is insufficient for nested virtualization case
- L1 vCPU executes different security domains at different times
  - L1 hypervisor will be context switching between L1 hypervisor and L2 guests
- However, L0 hypervisor maintains single VA space for the L1 VM
  - Which contains data for both L1 guest and L2 guests

# Hyper-V HyperClear: Sensitive Data Scrubbing

- L0 Hyper-V avoids caching any sensitive guest state in its data structures
- If L0 must read guest data into its private memory, clear it before exiting L0
  - E.g. Read vCPU registers on IOPort access emulation
- Ensures sensitive L1 memory not resident in cache when entering L2 context
- Note: L1D cache is flushed when switching between L1 and L2



Source:  
<https://techcommunity.microsoft.com/t5/Virtualization/Hyper-V-HyperClear-Mitigation-for-L1-Terminal-Fault/ba-p/382429>

# Summary

- L1TF SMT attack scenarios:
  - 1) Sibling running other VM vCPU  $\Leftarrow$  Mitigated by Core-Scheduler
  - 2) Sibling running #VMExit handler  $\Leftarrow$  Mitigated by reducing KVM VA space

# Summary

- L1TF SMT attack scenarios:
  - 1) Sibling running other VM vCPU  $\Leftarrow$  Mitigated by Core-Scheduler
  - 2) Sibling running #VMExit handler  $\Leftarrow$  Mitigated by reducing KVM VA space
- Reduce KVM VA space techniques:

XPFO, Process-local kernel VA region & **Kernel ASI**

  - **Kernel ASI doesn't require to flush L1D\$ on every VMEntry**

# Summary

- L1TF SMT attack scenarios:
  - 1) Sibling running other VM vCPU  $\Leftarrow$  Mitigated by Core-Scheduler
  - 2) Sibling running #VMExit handler  $\Leftarrow$  Mitigated by reducing KVM VA space
- Reduce KVM VA space techniques:

XPFO, Process-local kernel VA region & **Kernel ASI**

  - Kernel ASI doesn't require to flush L1D\$ on every VMEntry
- Kernel ASI **Decouples ASI VA space from host VA space**
- Kernel ASI is a **whitelist** approach



# Summary

- L1TF SMT attack scenarios:
  - 1) Sibling running other VM vCPU  $\Leftarrow$  Mitigated by Core-Scheduler
  - 2) Sibling running #VMExit handler  $\Leftarrow$  Mitigated by reducing KVM VA space
- Reduce KVM VA space techniques:

XPFO, Process-local kernel VA region & **Kernel ASI**

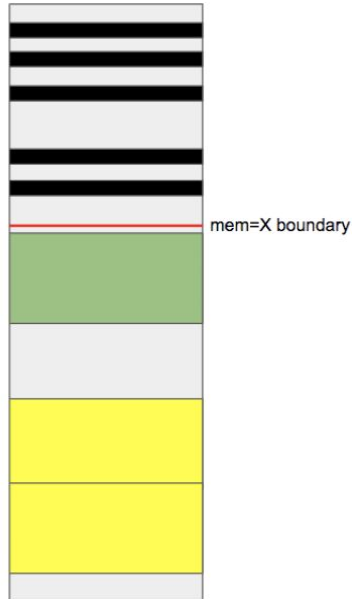
  - **Kernel ASI doesn't require to flush L1D\$ on every VMEntry**
- Kernel ASI **Decouples ASI VA space from host VA space**
- Kernel ASI is a **whitelist** approach
- **Kernel ASI only known method to mitigate MDS user $\rightarrow$ kernel SMT attack**

# Future Mitigations

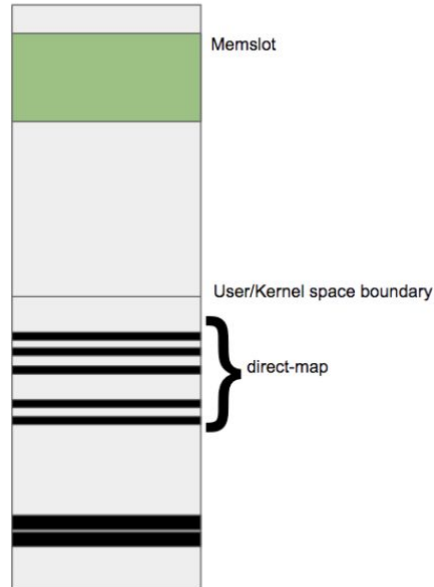
# mem=X

- mem=X is a boot option to limit amount of host DRAM used by kernel
- What if we will allocate physical pages for guest sensitive data from non-X?

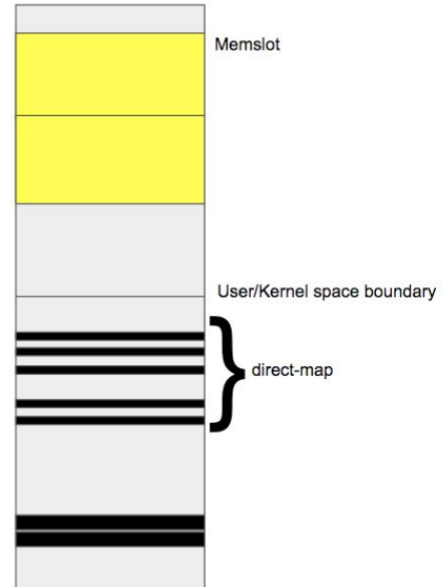
Physical  
Address Space



Guest 1  
Virtual Address Space



Guest 2  
Virtual Address Space



# mem=X: Security Analysis

- All guest sensitive data are not mapped in “direct-map”
  - Kernel does not map non-X physical pages in “direct-map”
- Map non-X physical pages in EPT & Process-local PTEs only
  - Process-local PTEs == PTEs mapping userspace & “process-local kernel VA region”
- Can also be used to hide selected host sensitive data from guests

Stored in non-X physical page  
⇒ Not leakable by VMs

# mem=X: Performance Advantages

- Avoid wasting host DRAM on “struct page” for memory dedicated to guests
  - E.g. A host with 768GB RAM use 12GB ( $=768\text{GB}/4\text{K} * 64\text{bytes}$ ) of it for “struct page”
- Improves guest performance
  - E.g. Guest memory always mapped as 1GB pages
- Prevent host from using DRAM dedicated to guest for host cache
  - E.g. pagecache, inode cache, slab cache
- Allows disable speculative execution mitigations that hurt performance
  - L1D\$/MDS flushes, IBRS/retpoline, KPTI

# mem=X: Issues

- Kernel cannot perform zero-copy / scatter-gather without “struct page”
  - **Such as vhost kernel backend**
  - vfio-pci devices work as usual
  - Upcoming patches by Joao Martins (Oracle)
- Non-trivial to move all sensitive info to non-X & local-process kernel VA region
  - E.g. vCPU thread stack pages

Conclusion

# Conclusion

- No perfect public KVM solution yet for L1TF/MDS SMT attack vector
  - Administrators should be provided with options that best suit their needs
- Kernel ASI useful as generic mitigation against arbitrary read primitive
- ASI & mem=X de-facto **modify KVM environment into a Type-1 hypervisor**
  - i.e. Decouple KVM execution environment from rest of host (dom0)
  - Easier to modify KVM execution environment than entire host execution environment
  - **Questions arise on the security posture of Type-2 hypervisors in general**
- KVM community should constantly review other hypervisors to gain insights
  - E.g. Microsoft Hyper-V HyperClear



# Questions?

Thank you!