Agenda

- Example of problematic workloads to virtualize efficiently that currently trigger frequent VM-Exits with upstream KVM
- Recap of the different kinds of speculative execution attacks and mitigations
  - Not about HT/SMT (orthogonal and not enough time)
- Benchmarks of the current (kernel v5.3) speculative execution mitigations on VMX and SVM
- Two proposals to micro optimize the KVM VM-Exits in the host
Hard to virtualize guest workloads

• The most effective way optimize the guest mode is to reduce the number of VM-Exits with:
  ➢ device-assignment for I/O with hardware devices (VFIO, IOMMU, VT-d, SR-IOV, ...)
    ✓ Network | storage (NVME/SSD/SCSI) | GPU | RDMA
    ✓ vhost-user-blk/scsi/net for virtualized I/O
• Some guest workloads will still flood KVM with VM-Exits, for example:
  ✓ Guest scheduling events on idle vCPUs
    ✓ cpuidle-haltpoll upstream guest idle governor makes this case a lesser concern
    ✓ It risks wasting CPU in guest mode if the host isn’t idle
  ✓ Guest high resolution timers
Guest scheduling events on idle vCPUs

```c
if (fork()) {
    while (n--) {
        read(pipe1[0], buf, 1);
        write(pipe2[1], buf, 1);
    }
    wait(NULL);
} else {
    while (n--) {
        write(pipe1[1], buf, 1);
        read(pipe2[0], buf, 1);
    }
}
```

```
perf kvm stat record -a sleep 1

HOT CAN_GET_HOT ????

<table>
<thead>
<tr>
<th>VM-EXIT</th>
<th>Samples</th>
<th>Samples%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSR_WRITE</td>
<td>605044</td>
<td>75.08%</td>
</tr>
<tr>
<td>HLT</td>
<td>199774</td>
<td>24.79%</td>
</tr>
<tr>
<td>EXTERNAL_INTERRUPT</td>
<td>494</td>
<td>0.06%</td>
</tr>
<tr>
<td>PREEMPTION_TIMER</td>
<td>297</td>
<td>0.04%</td>
</tr>
<tr>
<td>PENDING_INTERRUPT</td>
<td>290</td>
<td>0.04%</td>
</tr>
<tr>
<td>MSR_READ</td>
<td>8</td>
<td>0.00%</td>
</tr>
<tr>
<td>EPT_MISCONFIG</td>
<td>6</td>
<td>0.00%</td>
</tr>
<tr>
<td>PAUSE_INSTRUCTION</td>
<td>3</td>
<td>0.00%</td>
</tr>
</tbody>
</table>
```
Guest high resolution timers

```c
sigevent.sigev_notify = SIGEV_SIGNAL;
sigevent.sigev_signo = SIGALRM;
sigevent.sigev_value.sival_ptr = &timer;
if (timer_create(CLOCK_REALTIME, &sigevent, &timer) < 0)
    perror("timer_create"), exit(1);

itimerspec.it_value.tv_sec = 0;
itimerspec.it_value.tv_nsec = 1;
itimerspec.it_interval.tv_sec = 0;
itimerspec.it_interval.tv_nsec = 1;
if (timer_settime(timer, 0, &itimerspec, NULL) < 0)
    perror("timer_settime"), exit(1);

for(;;) pause();
```

```
perf kvm stat record -a sleep 1

<table>
<thead>
<tr>
<th>VM-EXIT</th>
<th>Samples</th>
<th>Samples%</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSR_WRITE</td>
<td>338793</td>
<td>56.54%</td>
</tr>
<tr>
<td>PENDING_INTERRUPT</td>
<td>168431</td>
<td>28.11%</td>
</tr>
<tr>
<td>PREEMPTION_TIMER</td>
<td>91723</td>
<td>15.31%</td>
</tr>
<tr>
<td>EXTERNAL_INTERRUPT</td>
<td>234</td>
<td>0.04%</td>
</tr>
<tr>
<td>HLT</td>
<td>65</td>
<td>0.01%</td>
</tr>
<tr>
<td>MSR_READ</td>
<td>6</td>
<td>0.00%</td>
</tr>
<tr>
<td>EPT_MISCONFIG</td>
<td>6</td>
<td>0.00%</td>
</tr>
</tbody>
</table>
```
Hard to virtualize guest workloads

- Some databases incidentally tend to be very heavy in terms of:
  - Frequent scheduling on potentially otherwise idle vCPU
  - Programming high frequency timers running at fairly high frequency
- Even an increase of 10% in the computation time of guest mode compared to bare metal can become quite problematic
  - Every 1% lost anywhere matters if the maximum you can lose is 10%
- Performance regressed for those hard to virtualize workloads since Jan 4 2018
  - "spectre-v2" default retpoline mitigation is important in the KVM host
  - "spec_store_bypass_disable=seccomp spectre_v2_user=seccomp" is still used as the guest default
Recap: 4 different attack targets

- Kernel Mode / KVM
- User Mode (user task)
- Guest Mode (user task)
- User Mode (user task)
- User Mode (user task)
- User Mode (user task)

Running tasks

CPU 0
- Core 0
- Runqueue

CPU 1
- Core 0
- Runqueue

CPU 2
- Core 1
- Runqueue

CPU 3
- Core 1
- Runqueue

Guest Mode (user task)

2 HT/SMT 1 CORE

Guest Mode (user task)

2 HT/SMT 1 CORE

User Mode (user task)

User Mode (user task)

User Mode (user task)

User Mode (user task)
Kernel attack (retpoline/IBRS/verw/PTI)

2 HT/SMT 1 CORE

Guest Mode (user task)
CPU 0 CORE 0 RUNQUEUE

User Mode (user task)
CPU 1 CORE 0 RUNQUEUE

Running tasks

User Mode (user task)

Guest Mode (user task)
CPU 2 CORE 1 RUNQUEUE

Guest Mode (user task)
CPU 3 CORE 1 RUNQUEUE

User Mode (user task)

Kernel Mode KVM

KVM
Context switch attack (IBPB/RSBfill)

Kernel Mode
KVM

Guest Mode (user task)
CPU 3
CORE 1
RUNQUEUE

User Mode (user task)
CPU 2
CORE 1
RUNQUEUE

Guest Mode (user task)
CPU 1
CORE 0
RUNQUEUE

User Mode (user task)
CPU 0
CORE 0
RUNQUEUE

User Mode (user task)
CPU 3
CORE 1
RUNQUEUE

Guest Mode (user task)
CPU 2
CORE 1
RUNQUEUE

User Mode (user task)
CPU 1
CORE 0
RUNQUEUE

Guest Mode (user task)
CPU 3
CORE 1
RUNQUEUE

User Mode (user task)
CPU 1
CORE 0
RUNQUEUE

Running tasks

2 HT/SMT
1 CORE

2 HT/SMT
1 CORE
HT/SMT attack (STIBP/nosmt/ASI)
Within-process JIT attack (SSBD)
Mitigations opt-outs

- For vulnerabilities that don’t require knowing the code that is running in the CPU:
  - Meltdown → pti=off
  - L1TF → l1tf=off
  - MDS → mds=off
  - FPU state and other registers → no turnoff

- For vulnerabilities that require knowing the code that is running in the CPU:
  - Spectre v1 (barrier_nospec/swapgs etc..) → nospectre_v1
  - Spectre v2 → spectre_v2=off (kernel & context switch & HT attack)
  - Spectre v2 → spectre_v2_user=off (HT attack only)
  - SSBD → spec_store_bypass_disable=off (within process attack on the JIT memory from the JITed code)

- Global turnoff for all: mitigations=off (>= RHEL7.7)
KVM impact of spectre-v2 mitigation

• The spectre-v2 attack on the kernel/KVM by default is mitigated with *retpolines*

• *retpolines* are the best performing mitigation available
  ✔ On some CPUs it’s a full fix
  ✔ On some CPUs “risk of an attack low”
    ✔ On those CPUs RHEL kernels inform you in the boot log that you can opt-in the full fix with *spectre_v2=ibrs*

• *kvm.ko* calls *kvm_intel.ko* or *kvm_amd.ko* at every *VM-Exit* multiple times through the *kvm_x86_ops* pointer to functions
  ✔ This was not optimal before, but it become slower with retpolines causing extra cost for each single invocation of the *kvm_x86_ops* virtual methods
KVM x86 sub-modules with *kvm_x86_ops*

- **Guest Mode**
  - VM-Exit
  - `struct kvm_x86_ops vmx_x86_ops`
  - `struct kvm_x86_ops svm_x86_ops`
  - `kvm.ko`
  - `kvm-intel.ko`
  - `kvm-amd.ko`

Copyright © 2019 Red Hat Inc.
hrtimer 1sec - top 10 retpolines - VMX

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcpu_enter_guest+772</td>
<td>397680</td>
</tr>
<tr>
<td>kvm_arch_vcpu_ioctl_run+263</td>
<td>198848</td>
</tr>
<tr>
<td>kvm_vcpu_ioctl+559</td>
<td>198801</td>
</tr>
<tr>
<td>do_vfs_ioctl+164</td>
<td>198793</td>
</tr>
<tr>
<td>ksioctl+96</td>
<td>198771</td>
</tr>
<tr>
<td>__x64_sys_ioctl+22</td>
<td>198736</td>
</tr>
<tr>
<td>do_syscall_64+89</td>
<td>197697</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>vmx_vcpu_run.part.88+358</td>
<td>1987697</td>
</tr>
<tr>
<td>vcpu_enter_guest+1689</td>
<td>132405</td>
</tr>
<tr>
<td>kvm_arch_vcpu_ioctl_run+263</td>
<td>131046</td>
</tr>
<tr>
<td>kvm_vcpu_ioctl+559</td>
<td>131043</td>
</tr>
<tr>
<td>do_vfs_ioctl+164</td>
<td>131043</td>
</tr>
<tr>
<td>ksioctl+96</td>
<td>131043</td>
</tr>
<tr>
<td>__x64_sys_ioctl+22</td>
<td>131043</td>
</tr>
<tr>
<td>do_syscall_64+89</td>
<td>131043</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip_emulated_instruction+48</td>
<td>131046</td>
</tr>
<tr>
<td>handle_wrmsr+85</td>
<td>131046</td>
</tr>
<tr>
<td>vcpu_enter_guest+423</td>
<td>131046</td>
</tr>
<tr>
<td>kvm_arch_vcpu_ioctl_run+263</td>
<td>131046</td>
</tr>
<tr>
<td>kvm_vcpu_ioctl+559</td>
<td>131046</td>
</tr>
<tr>
<td>do_vfs_ioctl+164</td>
<td>131046</td>
</tr>
<tr>
<td>ksioctl+96</td>
<td>131046</td>
</tr>
<tr>
<td>__x64_sys_ioctl+22</td>
<td>131046</td>
</tr>
<tr>
<td>do_syscall_64+89</td>
<td>131046</td>
</tr>
<tr>
<td>handle_wrmsr+102</td>
<td>131046</td>
</tr>
<tr>
<td>vcpu_enter_guest+772</td>
<td>131046</td>
</tr>
<tr>
<td>kvm_arch_vcpu_ioctl_run+263</td>
<td>131046</td>
</tr>
<tr>
<td>kvm_vcpu_ioctl+559</td>
<td>131046</td>
</tr>
<tr>
<td>do_vfs_ioctl+164</td>
<td>131046</td>
</tr>
<tr>
<td>ksioctl+96</td>
<td>131046</td>
</tr>
<tr>
<td>__x64_sys_ioctl+22</td>
<td>131046</td>
</tr>
<tr>
<td>do_syscall_64+89</td>
<td>131046</td>
</tr>
</tbody>
</table>
hrtimer 1sec - top 10 retpolines – SVM

16

vcpu_enter_guest+772
kvm_arch_vcpu_ioctl+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]
]: 227076

vcpu_enter_guest+168
kvm_arch_vcpu_ioctl+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]
]: 113601

vcpu_enter_guest+486
kvm_arch_vcpu_ioctl+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]
]: 113414

vcpu_enter_guest+423
kvm_arch_vcpu_ioctl+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]
]: 113386

vcpu_enter_guest+575
kvm_arch_vcpu_ioctl+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]
]: 113371

vcpu_enter_guest+1689
kvm_arch_vcpu_ioctl+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]
]: 112579

vcpu_enter_guest+4009
kvm_arch_vcpu_ioctl+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]
]: 75812

kvm_get_rflags+28
svm_interrupt_allowed+50
vcpu_enter_guest+4009
kvm_arch_vcpu_ioctl+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]
]: 75647

msr_interception+138
vcpu_enter_guest+772
kvm_arch_vcpu_ioctl+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]
]: 74795

msr_interception+356
vcpu_enter_guest+772
kvm_arch_vcpu_ioctl+263
kvm_vcpu_ioctl+559
do_vfs_ioctl+164
ksys_ioctl+96
__x64_sys_ioctl+22
do_syscall_64+89
]
]: 74757
KVM monolithic

• The objective is the retpoline elimination from VM-Exits
  ▶ remove the `kvm_x86_ops`
  ▶ remove `kvm.ko`
    • the executable `.o objects` previously linked in `kvm.ko` need to be duplicated and linked statically in **both** `kvm-intel.ko` and `kvm-amd.ko`

• pvops could eliminate retpolines, but they're suboptimal for iTLB (and RAM) costs

• Only two cons depending on `CONFIG_KVM_INTEL` and `CONFIG_KVM_AMD .config`:
  ▶ Only one of the two options can be set to “=*y*” **at once**
  ▶ Hint: distro kernels sets both “=*m*”
  ▶ If both set “=*m*”, a **few MiB of disk space** will be lost in `/lib/modules/"
KVM x86 sub-modules with *kvm_x86_ops*

- No benefit: *kvm-intel.ko* and *kvm-amd.ko* can’t be loaded at the same time
  - Because of hardware constraints
KVM monolithic (no \textit{kvm\_x86\_ops})

- Replace all \textit{kvm\_x86\_ops} methods with external calls with the same name, but implemented differently in \textit{kvm-intel.ko} and \textit{kvm-amd.ko}
- Link all \textit{kvm.ko} code into both \textit{kvm-intel.ko} and \textit{kvm-amd.ko}
KVM VM-Exit handler optimization

• *kvm_x86_ops* (and *kvm_pmu_ops*) aren’t the only sources of frequent *retpolines* during VM-Exits

• Unlike the *kvm_x86_ops*, invoking the VM-Exit reason handler *pointer to function* was optimal if the retpolines are not enabled
  
  ▶ In this case we can add the retpoline optimization conditional to `#ifdef CONFIG_RETPOLINE`
if (exit_reason < kvm_vmx_max_exit_handlers -
    && kvm_vmx_exit_handlers[exit_reason]) +
    && kvm_vmx_exit_handlers[exit_reason]) {
  #ifdef CONFIG_RETPOLINE
    if (exit_reason == EXIT_REASON_MSR_WRITE)
        return kvm_emulate_wrmsr(vcpu);
    else if (exit_reason == EXIT_REASON_PREEMPTION_TIMER)
        return handle_preemption_timer(vcpu);
    else if (exit_reason == EXIT_REASON_PENDING_INTERRUPT)
        return handle_interrupt_window(vcpu);
    else if (exit_reason == EXIT_REASON_EXTERNAL_INTERRUPT)
        return handle_external_interrupt(vcpu);
    else if (exit_reason == EXIT_REASON_HLT)
        return kvm_emulate_halt(vcpu);
    else if (exit_reason == EXIT_REASON_PAUSE_INSTRUCTION)
        return handle_pause(vcpu);
    else if (exit_reason == EXIT_REASON_MSR_READ)
        return kvm_emulate_rdmsr(vcpu);
    else if (exit_reason == EXIT_REASON_CPUID)
        return kvm_emulate_cpuid(vcpu);
    else if (exit_reason == EXIT_REASON_EPT_MISCONFIG)
        return handle_ept_misconfig(vcpu);
  #endif
  return kvm_vmx_exit_handlers[exit_reason](vcpu);
KVM VM-Exit handler optimization - SVM

```c
+#ifdef CONFIG_RETPOLINE
+
+    if (exit_code == SVM_EXIT_MSR)
+        return msr_interception(svm);
+    else if (exit_code == SVM_EXIT_VINTR)
+        return interrupt_window_interception(svm);
+    else if (exit_code == SVM_EXIT_INTR)
+        return intr_interception(svm);
+    else if (exit_code == SVM_EXIT_HLT)
+        return halt_interception(svm);
+    else if (exit_code == SVM_EXIT_NPF)
+        return npf_interception(svm);
+    else if (exit_code == SVM_EXIT_CPUID)
+        return cpuid_interception(svm);
+    +#endif
+
+    return svm_exit_handlers[exit_code](svm);
```
hrtimer 1sec - top 5 retpolines – VMX - after

\[
\begin{align*}
\_kvm\_wait\_lapic\_expire+284 & \quad \text{do_syscall\_64+89} \\
vmx\_vcpu\_run.part.97+1091 & \\
vcpu\_enter\_guest+377 & \\
kvm\_arch\_vcpu\_ioctl\_run+261 & \\
kvm\_vcpu\_ioctl+559 & \\
do\_vfs\_ioctl+164 & \\
ksys\_ioctl+96 & \\
\_x64\_sys\_ioctl+22 & \\
do\_syscall\_64+89 & : 2390 \\
\end{align*}
\]

\[
\begin{align*}
\text{finish\_task\_switch+371} & \\
\_schedule+573 & \\
preempt\_schedule\_common+10 & \\
\_cond\_resched+29 & \\
kvm\_arch\_vcpu\_ioctl\_run+401 & \\
kvm\_vcpu\_ioctl+559 & \\
do\_vfs\_ioctl+164 & \\
ksys\_ioctl+96 & \\
\_x64\_sys\_ioctl+22 & \\
do\_syscall\_64+89 & : 103 \\
\end{align*}
\]

\[
\begin{align*}
\_schedule+1081 & \\
preempt\_schedule\_common+10 & \\
\_cond\_resched+29 & \\
kvm\_arch\_vcpu\_ioctl\_run+401 & \\
kvm\_vcpu\_ioctl+559 & \\
do\_vfs\_ioctl+164 & \\
ksys\_ioctl+96 & \\
\_x64\_sys\_ioctl+22 & \\
do\_syscall\_64+89 & : 57 \\
\end{align*}
\]

\[
\begin{align*}
ktime\_get\_update\_offsets\_now+70 & \\
hrtimer\_interrupt+131 & \\
sm\_apic\_timer\_interrupt+106 & \\
apic\_timer\_interrupt+15 & \\
vcpu\_enter\_guest+1119 & \\
kvm\_arch\_vcpu\_ioctl\_run+261 & \\
kvm\_vcpu\_ioctl+559 & \\
do\_vfs\_ioctl+164 & \\
ksys\_ioctl+96 & \\
\_x64\_sys\_ioctl+22 & \\
do\_syscall\_64+89 & \\
\end{align*}
\]

\[
\begin{align*}
delay\_fn() & \\
\_delay & \\
\end{align*}
\]
hrtimer 1sec - top 5 retpolines – SVM - after

```c
ktime_get+58
  start_sw_timer+279
  restart_apic_timer+85
  kvm_set_msr_common+1497
  msr_interception+142
  vcpu_enter_guest+684
  kvm_vcpu_ioctl+559
  do_vfs_ioctl+164
  ksys_ioctl+96
  __x64_sys_ioctl+22
  do_syscall_64+89
}: 499845
```

```c
ktime_get+58
  clockevents_program_event+84
  hrtimer_try_to_cancel+168
  hrtimer_cancel+21
  kvm_set_lapic_tscdeadline_msr+43
  kvm_set_msr_common+1497
  msr_interception+142
  vcpu_enter_guest+684
  kvm_arch_vcpu_ioctl_run+261
  kvm_vcpu_ioctl+559
  do_vfs_ioctl+164
  ksys_ioctl+96
  __x64_sys_ioctl+22
  do_syscall_64+89
}: 42848
```

```c
clockevents_program_event+148
  hrtimer_try_to_cancel+168
  hrtimer_cancel+21
  kvm_set_lapic_tscdeadline_msr+43
  kvm_set_msr_common+1497
  msr_interception+142
  vcpu_enter_guest+684
  kvm_arch_vcpu_ioctl_run+261
  kvm_vcpu_ioctl+559
  do_vfs_ioctl+164
  ksys_ioctl+96
  __x64_sys_ioctl+22
  do_syscall_64+89
}: 42766
```

```c
lapic_next_event+28
  clockevents_program_event+148
  hrtimer_try_to_cancel+168
  hrtimer_cancel+21
  kvm_set_lapic_tscdeadline_msr+43
  kvm_set_msr_common+1497
  msr_interception+142
  vcpu_enter_guest+684
  kvm_arch_vcpu_ioctl_run+261
  kvm_vcpu_ioctl+559
  do_vfs_ioctl+164
  ksys_ioctl+96
  __x64_sys_ioctl+22
  do_syscall_64+89
}: 42723
```

```c
ktime_get+58
  clockevents_program_event+84
  hrtimer_start_range_ns+528
  start_sw_timer+356
  restart_apic_timer+85
  kvm_set_msr_common+1497
  msr_interception+142
  vcpu_enter_guest+684
  kvm_arch_vcpu_ioctl_run+261
  kvm_vcpu_ioctl+559
  do_vfs_ioctl+164
  ksys_ioctl+96
  __x64_sys_ioctl+22
  do_syscall_64+89
}: 41887
```

```c
ktime_get()
ktime_get()
apic->write();
ktime_get()
```
Micro benchmark disclaimer

- The following slides are going to show only **micro** benchmarks
- All **micro** benchmarks are **not representative** of useful or **real life workloads** or real life software applications or real life software products
- The results shown in the next slides **should not be taken at face value** and they don’t represent the real impact of the software mitigations against speculative execution side channel attacks
- All real and useful software applications running on Linux will show completely different benchmark results (i.e. a **much lower impact**) than what is shown in this slide deck
- **Micro** benchmarks in this slide deck are provided with the only purpose of
  - explaining how the software mitigation works
  - **justifying** to the community the KVM monolithic optimization developments or/and equivalent **software optimizations**
1 million CPUID loop

• Only useful to measure the VM-Exit latency

for (i=0; i < 10000000; i++)
    asm volatile("cpuid"
               : "=a" (eax),
                  "=b" (ebx),
                  "=c" (ecx),
                  "=d" (edx)
               : "0" (eax), "2" (ecx)
               : "memory");
Why l1tf=off is slower than l1tf=flush?

• This happens without MDS_NO and without mds=off and with MD_CLEAR set in the host cpuid
  ➢ The l1flush implies verw, but is conditional by default:
    ✔ After l1tf=off KVM executes verw at every VM-Enter
• CPUs with RCDL_NO always behave like l1tf=off by default
  ➢ Should still be faster than l1tf=cond?
skylake - upstream 5.3.0 KVM nosmt - 1 million cpuid loop

Guest mitigation settings

- 7.6% default
- spectre_v2=off
  - spec_store_bypass_disable=off
  - +1tf=off
- -20.2%
- -20.5%
- 0.0%
skylake - upstream 5.3.0 KVM nosmt - 1 million cpuid loop

Guest mitigation settings

- host_default: 35.4% default
- spectre_v2=off: 0.4%
- ..+spec_store_bypass_disable=off: 0.1%
- ..+11tf=off: 25.8%
- ..+mds=off: 0.0%
Guest mitigation settings
Mitigation **opt-out** with the most effect on VM-Exits: *spectre_v2=off*

i.e. optimizing away retpolines
Opt-in options are marked with “/”

Unlikely to be helpful (and spectre_v2=ibrs is missing on upstream 5.3.0)
Opt-in options are marked with "/"
Unlikely to be helpful (and spectre_v2=ibrs is missing on upstream 5.3.0)
skylake - upstream 5.3.0 KVM nosmt - 1 million cpuid loop

Guest mitigation settings

<table>
<thead>
<tr>
<th>Setting</th>
<th>Computation Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>l1tf=full</td>
<td>0.77</td>
</tr>
<tr>
<td>spec_store_bypass_disable=on</td>
<td>0.33</td>
</tr>
<tr>
<td>host_default</td>
<td>0.01</td>
</tr>
<tr>
<td>spectre_v2=off</td>
<td>0.24</td>
</tr>
<tr>
<td>...spec_store_bypass_disable=off</td>
<td>0.24</td>
</tr>
<tr>
<td>...l1tf=off</td>
<td>0.24</td>
</tr>
<tr>
<td>...mds=off</td>
<td>0.24</td>
</tr>
<tr>
<td>...pti=off</td>
<td>0.24</td>
</tr>
<tr>
<td>...nospectre_v1</td>
<td>0.24</td>
</tr>
</tbody>
</table>

© 2019 Red Hat Inc.
Why ssbd=off in guest speed up KVM?

- `spec_store_bypass_disable=auto` is the guest and host default
  - In practice the default is `spec_store_bypass_disable=seccomp`
- NOTE: the guest cpuid loop doesn't run under SECCOMP
- Problem: nearly everything else nowadays occasionally uses SECCOMP (sshd etc..)
- The first use of SECCOMP in guest will write SSBD to `SPEC_CTRL` and *will forever slowdown the guest*
  - `rdmsr(SPEC_CTRL)` forced at every VM-Exit
spectre_v2=off was already set

- `spectre_v2_user=off` was already implied by `spectre_v2=off` at the previous step
  - If not disabling `spectre_v2_user` too, `SPEC_CTRL` may still be written to for `STIBP`

- `spectre_v2_user=auto` is the upstream default
  - With most distro `.config` the default is `spectre_v2_user=seccomp`
ssbd = spectre_v2_user = seccomp

- \texttt{seccomp} was a good default on un-embargo day (as a “catch-all”)
- \texttt{seccomp} looks too coarse by now
  - Doesn’t only hurt the guest performance
  - It hurts all SECCOMP users even on bare metal
    - More and more software runs under SECCOMP including Kubernetes pods and podman containers
- \texttt{prctl} would be a preferred default now because the apps who need STIBP or SSBD should have added \texttt{prctl(PR\_SET\_SPECULATION\_CTRL)}
- There has never been a \textit{guarantee} that code requiring STIBP or SSBD runs under SECCOMP in the first place
- SECCOMP users are adding \texttt{SECCOMP\_FILTER\_FLAG\_SPEC\_ALLOW} to their userland as band-aid for the too coarse default that slowdown SECCOMP on \textit{bare metal}
SSBD spec_store_bypass_disable=prctl

• Who really needs to set SSBD?
  ➢ JIT running un-trusted bytecode (i.e. desktop usage of javaws/applet)
    ➢ to avoid the JITed code to read the “in-process” memory of the JIT
• We patched OpenJDK JIT downstream with the prctl() during the SSBD embargo
• Upstream OpenJDK makes no guarantees of "in-process" data confidentiality
  ➢ The prctl() was never submitted to OpenJDK
    ➢ If not even the JIT enforces SSBD, why all non-JIT SECCOMP users should?
• Overall the prctl() should be worth it if the javaws/applet runs with reduced permissions
• SSBD provides no benefits after a privilege escalation that takes over the code running in any SECCOMP jail
  ➢ After privilege escalation any malicious code can read the memory of the thread regardless if SSBD is set or not
  ➢ Bad fit for the SECCOMP model
STIBP spectre_v2_user=prctl

• In theory STIBP is a good fit for the SECCOMP model
• In practice the spectre-v2 HT attack prevented by STIBP would require to know 1) the code and 2) the virtual address it is running at in the other hyper-thread
• Security sensitive code jailed under SECCOMP would better run also under \textit{PID namespaces/VM isolation}
  ➢ If so the jailed code \textit{can’t know what’s running} in the other hyper-thread
• Even without PID namespaces/VM isolation the address is it running at is \textit{randomized with ASLR}
STIBP spectre_v2_user=prctl

- **Since MDS** very few CPUs are immune from MDS without *nosmt*
  - Which would render **STIBP irrelevant** by disabling HT
- MDS can be attacked with HT enabled even without knowing what’s running in the other CPU and the virtual address it is running at
  - PID namespaces and ASLR won’t help with MDS
- STIBP mitigates *spectre-v2-HT* from a SECCOMP jail that **could still be able to exploit MDS**
- Even retpolines aren’t a full fix on some CPU, yet they’re the default upstream
  - A spectre-v2 attack against the *kernel* would have a more spread spectrum than a spectre-v2 attack against *HT*
Guest mitigation settings

- guest_default
- spectre_v2=off
- spec_store_bypass_disable=off
- +spec_store_bypass_disable=off
- +1tf=off
- +mds=off
- +pti=off
- +nospectre_v1
KVM monolithic

20.7% difference in compute time

*default* mitigations in guest and host
skylake - KVM monolithic v2 on 5.3.0 nosmt - 1 million cpuid loop

KVM monolithic + spec_store_bypass_disable=prctl + spectre_v2_user=prctl = 57.3% difference in compute time
Mitigation opt-out with the most effect on VM-Exits: `spectre_v2=off` i.e. optimizing away retpolines

epyc - upstream 5.3.0 KVM nosmt - 1 million cpuid loop
KVM monolithic

13.2% difference in compute time
default mitigations in guest and host
KVM monolithic

4.1% difference in compute time even with *mitigations=off*
skylake - KVM monolithic v2 on 5.3.0 nosmt - 1 proc raising 1 million SIGALRM with setitimer(1nsec)

KVM monolithic

5.8% difference in compute time

default mitigations in guest and host
epyc - KVM monolithic v2 on 5.3.0 nosmt - 1 proc raising 1 million SIGALRM with setitimer(1nsec)

KVM monolithic

7.6% difference in compute time

default mitigations in guest and host
KVM monolithic status

- *KVM monolithic kernel patch-set posted on kvm@ and lkml@
  https://lkml.kernel.org/r/20190928172323.14663-1-aarcange@redhat.com*

- Kbuild options need more adjustment
- Some warnings from duplicated exports
- Final cleanup of *kvm_x86_ops* pending because it can be done incrementally (cleaner)
  - *kvm_pmu_ops* already removed
Thank you

Red Hat is the world’s leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.