Inter-VM data exfiltration

The art of cache timing covert channel on x86 multi-core

Etienne Martineau
Kernel Developer

August 2015
Disclaimer

- Research… own time… my opinions… not my employers…

- The information and the code provided in this presentation is to be used for educational purposes only.

- I am in no way responsible for any misuse of the information provided.

- In no way should you use the information to cause any kind of damage directly or indirectly.
About me

\[ \nabla \cdot \mathbf{E} = \frac{\rho}{\varepsilon_0} = 4\pi k \rho \]
\[ \nabla \cdot \mathbf{B} = 0 \]
\[ \nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \]
\[ \nabla \times \mathbf{B} = \frac{\mathbf{j}}{\varepsilon_0 c^2} + \frac{1}{c^2} \frac{\partial \mathbf{E}}{\partial t} \]
\[ \int \mathbf{E} \cdot d\mathbf{A} = \frac{q}{\varepsilon_0} \]
\[ \int \mathbf{B} \cdot d\mathbf{A} = 0 \]
\[ \int \mathbf{E} \cdot d\mathbf{s} = -\frac{d\Phi_B}{dt} \]
\[ \int \mathbf{B} \cdot d\mathbf{s} = \mu_0 i + \frac{1}{c^2} \frac{\partial}{\partial t} \int \mathbf{E} \cdot d\mathbf{A} \]
<table>
<thead>
<tr>
<th>Hyper-threaded</th>
<th>Non Hyper-threaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>vCPU0</td>
<td>vCPU1</td>
</tr>
<tr>
<td>200000</td>
<td>200204</td>
</tr>
<tr>
<td>200088</td>
<td>200084</td>
</tr>
<tr>
<td>210768</td>
<td>193512</td>
</tr>
<tr>
<td>200096</td>
<td>200084</td>
</tr>
<tr>
<td>200072</td>
<td>200100</td>
</tr>
<tr>
<td>187312</td>
<td>226556</td>
</tr>
<tr>
<td>204776</td>
<td>205364</td>
</tr>
<tr>
<td>186996</td>
<td>231952</td>
</tr>
<tr>
<td>200016</td>
<td>200176</td>
</tr>
<tr>
<td>200088</td>
<td>200084</td>
</tr>
<tr>
<td>200084</td>
<td>200088</td>
</tr>
<tr>
<td>200076</td>
<td>200096</td>
</tr>
<tr>
<td>200084</td>
<td>200088</td>
</tr>
<tr>
<td>200240</td>
<td>191980</td>
</tr>
<tr>
<td>204588</td>
<td>205536</td>
</tr>
<tr>
<td>Hyper-threaded</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>--</td>
</tr>
<tr>
<td>vCPU0</td>
<td>vCPU1</td>
</tr>
<tr>
<td>200000</td>
<td>200228</td>
</tr>
<tr>
<td>200088</td>
<td>200084</td>
</tr>
<tr>
<td>210768</td>
<td>193512</td>
</tr>
<tr>
<td>200096</td>
<td>200084</td>
</tr>
<tr>
<td>200072</td>
<td>200100</td>
</tr>
<tr>
<td>187312</td>
<td>226556</td>
</tr>
<tr>
<td>204776</td>
<td>205364</td>
</tr>
<tr>
<td>186996</td>
<td>231952</td>
</tr>
<tr>
<td>200016</td>
<td>200176</td>
</tr>
<tr>
<td>200088</td>
<td>200084</td>
</tr>
<tr>
<td>200084</td>
<td>200088</td>
</tr>
<tr>
<td>200076</td>
<td>200096</td>
</tr>
<tr>
<td>200084</td>
<td>200088</td>
</tr>
<tr>
<td>200240</td>
<td>191980</td>
</tr>
<tr>
<td>204588</td>
<td>205364</td>
</tr>
<tr>
<td>200000</td>
<td>200204</td>
</tr>
</tbody>
</table>

![Diagram of Core and VM Relationships](image-url)
An Intel processor with HT Technology can execute two software threads in an increasingly parallel manner, utilizing previously unused resources.
VM#1  Modulate a contention pattern
1 | 0 | 0 | 0 | 1
MUL | NOP | NOP | NOP | MUL
**VM#1** Modulate a contention pattern

1 | 0 | 0 | 0 | 1

MUL | NOP | NOP | NOP | MUL

**VM#2** Detect BUS contention

Slow | Fast | Fast | Fast | Slow

1 | 0 | 0 | 0 | 1
Video #1
Overview

- Goal
  - Practical implementation (not just some research stuff)

- How
  - Abusing X86 shared resources
  - Cache line encoding/decoding
  - Getting around the HW pre-fetcher
  - Data persistency and noise. What can be done?
  - Guest to host page table de-obfuscation. The easy way
  - High precision inter-VM synchronization: All about timers

- Detection / Mitigation
Shared resource: HT enabled

- Pipeline contention
- "previous example"
- L1 modulation
- L2 modulation

VM #1

VM #2
Shared resource: HT disabled

http://it.slashdot.org/story/05/05/17/201253/hyper-threading-linus-torvalds-vs-colin-percival
Shared resource: Multi socket

VM #1
Core #0
Core #1
Core #2
Core #3
VM #2
Cache line (64 bytes)
byte

Cache line (64 bytes)
VM#1 encode a pattern in cache line

<table>
<thead>
<tr>
<th>CL0</th>
<th>CL1</th>
<th>CL2</th>
<th>CL3</th>
<th>CL4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Load | Flush | Flush | Flush | Load
**VM#1** encode a pattern in cache line

<table>
<thead>
<tr>
<th>CL0</th>
<th>CL1</th>
<th>CL2</th>
<th>CL3</th>
<th>CL4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Load | Flush| Flush| Flush| Load

**VM#2** decode the cache line access time

<table>
<thead>
<tr>
<th>CL0</th>
<th>CL1</th>
<th>CL2</th>
<th>CL3</th>
<th>CL4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast</td>
<td>Slow</td>
<td>Slow</td>
<td>Slow</td>
<td>Fast</td>
</tr>
</tbody>
</table>

| 1   | 0   | 0   | 0   | 1   |
- NO VM
- Simple Client / Server test program
- Cache Line from shared memory directly
- Mutex for inter-process signaling
- Client encode a pattern
- NO VM
- Simple Client / Server test program
- Cache Line from shared memory directly
- Mutex for inter-process signaling
- Client encode a pattern
- Server decode
- ➔ Something weird?
• Simple test:
• Flush CL0 -> CL100
• Measure CL access time for CL0 -> CL100
• ➔ Long latency for all CL
• Simple test:
• Flush CL0 -> CL100
• Measure CL access time for CL0 -> CL100
• ➔ Long latency for all CL
• ???
Prefetching in general means bringing data or instructions from memory into the cache **before they are needed**
Prefetching in general means bringing data or instructions from memory into the cache **before they are needed**

The Core™ i7 processor and Xeon® 5500 series processors, for example, have some prefetchers that bring data into the L1 cache and some that bring data into the L2.

There are also different algorithms – some monitor data access patterns for a particular cache and then try to predict what addresses will be needed in the future.

- Simple test:
  - Flush CL0 -> CL100
  - Measure CL access time for CL0 -> CL100
  - ➔ Long latency for all CL

  ```
  Zap Cache Line 0->100: DONE
  Load Cache Line 0->100 ( TSC cycle ):
  240 264 232 232 236 232 232 228 232 232
  236 228 68 68 64 68 232 260 232 232
  232 232 232 232 232 232 236 232 64 64
  64 64 64 64 64 68 64 64 64 68
  64 64 64 64 68 64 64 68 68 64
  64 64 64 64 64 68 64 68 64 68
  64 64 64 64 64 64 64 64 64 64
  232 236 232 236 236 228 232 228 236
  64 64 64 64 64 64 64 64 64 64
  64 64 64 64 64 64 64 64 64 64
  64 64 64 64 64 64 64 64 64 64
  64 68 64 64 64 64 64 64 64 64
  64 68 64 64 64 64 64 64 64 64
  ```
Prefetching in general means bringing data or instructions from memory into the cache before they are needed.

The Core™ i7 processor and Xeon® 5500 series processors, for example, have some prefetchers that bring data into the L1 cache and some that bring data into the L2.

There are also different algorithms – some monitor data access patterns for a particular cache and then try to predict what addresses will be needed in the future.

- Simple test:
- Flush CL0 -> CL100
- Measure CL access time

<table>
<thead>
<tr>
<th>Load Cache Line 0-&gt;100 ( TSC cycle):</th>
</tr>
</thead>
<tbody>
<tr>
<td>240 264 232 232 236</td>
</tr>
<tr>
<td>236 228 68 68 64</td>
</tr>
<tr>
<td>232 232 232 232 232</td>
</tr>
<tr>
<td>64 64 64 64 64</td>
</tr>
<tr>
<td>64 64 64 64 64</td>
</tr>
<tr>
<td>64 64 64 64 64</td>
</tr>
<tr>
<td>64 64 68 64 64 64 232</td>
</tr>
<tr>
<td>232 236 232 232 232 228</td>
</tr>
<tr>
<td>64 64 64 64 64</td>
</tr>
<tr>
<td>64 64 64 64 64</td>
</tr>
</tbody>
</table>

**PCI Parity:**
- Plug and Play Operating System: [Enabled]
- Legacy Free: [No]
- Default Primary Video Adapter: [Auto]
- Turbo Memory: [Enabled]

**4GB PCI Hole (<1GB):** [512 MB]
**4GB PCI Hole Granularity:** [1.0 GB]
**Active Processors:** [Max. Cores]
**Hyperthreading:** [Disabled]
**Set Max Ext CPUID = 3**
**Hardware Prefetcher:** [Enabled]
**Adjacent Cache Line Prefetch**
**Execute Disable Bit:** [Enabled]
**Intel(R) Virtualization Technology** [Enabled]
- Simple trick that randomized CL access
- Simple trick that randomized CL access
- CL access random within a page
- Simple trick that randomized CL access
- CL access random within a page
- CL access random across pages
- Simple trick that randomized CL access
- CL access random within a page
- CL access random across pages
- This apparently manage to confound the HW prefetcher!
What happen if we wait longer before decoding?
What happen if we wait longer before decoding?

Wait
What happens if we wait longer before decoding?

- Wait
- Wait
- What happen if we wait longer before decoding?
- Wait
- Wait
- Wait
What happens if we wait longer before decoding?

- Wait
- Wait
- Wait

Encoded data in the cache evaporates pretty quickly.
Noise
Noise

Bare metal Kernel

Bare metal User
Noise
Noise

![Graphs showing CPU usage for different kernels and users in both bare metal and VM environments.](image)
- Client in **VM#1**, Server in **VM#2**
- Client in **VM#1**, Server in **VM#2**

- L2 OR L3 cache are tagged by the physical address but **in a VM the physical address that you see has nothing to do with the real physical address on bare metal that the cache is using.**
- Client in **VM#1**, Server in **VM#2**

- L2 OR L3 cache are tagged by the physical address but **in a VM the physical address that you see has nothing to do with the real physical address on bare metal that the cache is using.**

- There is another layer of translation
- Client in VM#1, Server in VM#2
- L2 OR L3 cache are tagged by the physical address but in a VM the physical address that you see has nothing to do with the real physical address on bare metal that the cache is using.
- There is another layer of translation
- This is a complex problem to solve
Page de-duplication

KSM enables the kernel to examine two or more already running programs and compare their memory. If any memory regions or pages are identical, **KSM merge them into a single page physical page on bare-metal host kernel.**
Page de-duplication

KSM enables the kernel to examine two or more already running programs and compare their memory. If any memory regions or pages are identical, **KSM merge them into a single page physical page on bare-metal host kernel.**

If one of the programs wants to modify a shared page KSM kicks in and un-merge it.
Page de-duplication

KSM enables the kernel to examine two or more already running programs and compare their memory. If any memory regions or pages are identical, KSM merge them into a single page physical page on bare-metal host kernel.

If one of the programs wants to modify a shared page KSM kicks in and un-merge it.

This is useful for virtualization with KVM. Once the guest is running the contents of the guest operating system image can be shared when guests are running the same operating system or applications.
Page table de-obfuscation
Page table de-obfuscation

The idea is to create a per-page unique pattern in memory that is the same across client and server
- Page table de-obfuscation
- The idea is to create a per-page unique pattern in memory that is the same across client and server
- So that on host KSM kicks in and do the page de-duplication for us
There is no synchronization primitive across processes running in different VMs?
- There is no synchronization primitive across processes running in different VMs.

- In reality there is a mechanism to do that (EX ivshmem) but this is not enabled in production env.
There is no synchronization primitive across processes running in different VMs.

In reality there is a mechanism to do that (e.g., ivshmem) but this is not enabled in production environments.

We need something to replace the mutex.
Option #1

- Forget about the synchronization aspect and hope for the best
- With error correction we can achieve some data transmission.
- Very low bit rates
- CPU consumption is low
Option #2

- Busy loop on each side
- Client faster than Server
- At some point there will be an overlap and the server will pickup the signal
- CPU consumption is High
- OK bit rates
- We want <1% CPU usage to remain undetected.
Option #3

- Define a common period ‘T’
- Client-Server lock into phase
Option #3

- Define a common period ‘T’
- Client-Server lock into phase
- Server sends a sync pattern
Option #3

- Define a common period ‘T’
- Client-Server lock into phase
- Server sends a sync pattern
- Client sweep over the period in search for the sync
Option #3

- Once the sync is found the phase is adjusted are we are ready for transmission.
Option #3

- Once the sync is found the phase is adjusted are we ready for transmission?

- For that to work we need a monotonic pulse
Option #3

- Once the sync is found the phase is adjusted are we are ready for transmission.
- For that to work we need a monotonic pulse
- Some jitter but not too much (Lots of noise in VMs ➔ data evaporates out of the cache very quickly)
How to achieve a monotonic pulse?
- How to achieve a monotonic pulse?
- Timers
How to achieve a monotonic pulse?

Timers

Why timers?

We need to sleep → Avoid detection ( < 1% CPU usage )
- How to achieve a monotonic pulse?
- Timers
- Why timers?
- We need to sleep ➔ Avoid detection ( < 1% CPU usage )
- Jitter comes from both VMs.
- Jitter comes from both VMs
- Too much jitter
- The idea here is to do padding up to some value above the maximum jitter.
- The idea here is to do padding up to some value above the maximum jitter
The idea here is to do padding up to some value above the maximum jitter.

The problem here is that the padding is subject to noise.

In other words, more time you spend trying to immunize yourself to noise, more noise you end up accumulating.
The idea here is to do padding up to some value above the maximum jitter.

The problem here is that the **padding is subject to noise**.

In other word more time you spend trying to immunize yourself to noise more noise you end up accumulating.

Padding consume CPU.

By stretching the timer period it’s easy to stay under 1% of CPU usage.
It’s a tricky problem but at the end I got it right!
- It’s a tricky problem but at the end I got it right!

- In short the padding is using a calibrated software loop that is kept in check with the TSC
It’s a tricky problem but at the end I got it right!

In short the padding is using a calibrated software loop that is kept in check with the TSC

Assume 2.4Ghz machine;

On a idle system: 
~50 cycle ➞ 20 nSec
It’s a tricky problem but at the end I got it right!

In short the padding is using a calibrated software loop that is kept in check with the TSC

Assume 2.4Ghz machine;

On a idle system:
~50 cycle  ➔  20 nSec

On a loaded system
~300 cycle  ➔  120 nSec
It’s a tricky problem but at the end I got it right!

In short the padding is using a calibrated software loop that is kept in check with the TSC

Assume 2.4Ghz machine;

On a idle system:
~50 cycle ➔ 20 nSec

On a loaded system
~300 cycle ➔ 120 nSec

Timers:
100uSec = 240 000 cycle
10uSec = 24 000 cycle (best case)
Recap

- Encoding / decoding based on memory access time
  - \((1 = \text{slow}, \, 0 = \text{fast})\)

- Got rid of the HW prefetching (without disabling it from BIOS!)
  - \((\text{randomized the access to cache lines/pages})\)

- Physical memory pages that are shared across VM
  - Thanks to KSM 😊

- PLL and high precision inter-VM synchronization
  - \((\text{Compensated timer} < 120 \text{ nSec jitter})\)

- Time for a demo!
Video #2
Video #3
Mitigation

- Disable page-deduplication (KSM) / Per-VM policy
  - No inter-VM shared read-only pages
  - Flush ‘clflush’ and reload won’t work
  - No OS / Application fingerprinting (de-duplication page-fault)
  - Higher memory cost

- X86 ‘clflush’ instruction: Privilege?
  - Microcode?

- Co-location policy (per-core / per-socket / per-box)
Detection

- Hardware counter
- Inter-VM scheduling “abnormality”
- TSC related “abnormality”
Thank you!